

Name: _____

This document contains some extra questions to give you some extra practice on some of the topics we discussed.

1. **Pipelines, Superscalar, and Scheduling I.** Below is the code for calculates the sum of the square of the difference of two arrays. The assembly is a simple RISC-like ISA that uses condition codes for branch outcomes. The BRn instruction means “branch if negative”.

```
loop:
  LD   R4, R1, 0    ; R4 <- Mem[R1+0]
  LD   R5, R2, 0    ; R5 <- Mem[R2+0]
  SUB  R6, R4, R5    ; R6 <- R4 - R5
  MUL  R7, R6, R6    ; R7 <- R6 * R6
  ADD  R3, R3, R7    ; R3 <- R3 + R7
  ADD  R1, R1, 1     ; R1 <- R1 + 1
  ADD  R2, R2, 1     ; R2 <- R2 + 1
  ADD  R0, R0, 1     ; R0 <- R0 + 1
  CMP  R0, 20
  BRn  loop
```

For the questions below, consider a 5-stage pipeline with a single cycle load-to-use penalty. All other instructions execute in a single cycle. All branches are predicted perfectly. For the superscalar pipelines assume any mix of instructions is allowed in a given cycle (for example, no limit on the number of loads per cycle), and because the BRn reads the condition codes set by CMP, BRn and CMP are not allowed to execute in the same cycle.

- (a) When executing the above code on a scalar (single-issue) pipeline, how many cycles per loop iteration? What is the CPI?

Answer: [3 points] 11 cycles per iteration, CPI = 1.1

- (b) When executing the above code on a two-way (dual-issue) superscalar pipeline, how many cycles per loop iteration? What is the CPI?

Answer: [2 points] 8 cycles per iteration, CPI = .8

- (c) Reschedule the code to achieve the best CPI on this dual-issue pipeline. Instead of rewriting the code, just fill in the cycle the instruction² begins execution in the “___” in the code below.

loop:

```
___ LD   R4,  R1,  0    ; R4 <- Mem[R1+0]
___ LD   R5,  R2,  0    ; R5 <- Mem[R2+0]
___ SUB  R6,  R4,  R5   ; R6 <- R4 - R5
___ MUL  R7,  R6,  R6   ; R7 <- R6 * R6
___ ADD  R3,  R3,  R7   ; R3 <- R3 + R7
___ ADD  R1,  R1,  1    ; R1 <- R1 + 1
___ ADD  R2,  R2,  1    ; R2 <- R2 + 1
___ ADD  R0,  R0,  1    ; R0 <- R0 + 1
___ CMP  R0,  20
___ BRn  loop
```

Answer: [5 points]

3

loop:

```
_1_ LD   R4, R1, 0    ; R4 <- Mem[R1+0]
_1_ LD   R5, R2, 0    ; R5 <- Mem[R2+0]
_3_ SUB  R6, R4, R5   ; R6 <- R4 - R5
_4_ MUL  R7, R6, R6   ; R7 <- R6 * R6
_5_ ADD  R3, R3, R7   ; R3 <- R3 + R7
_2_ ADD  R1, R1, 1    ; R1 <- R1 + 1
_2_ ADD  R2, R2, 1    ; R2 <- R2 + 1
_3_ ADD  R0, R0, 1    ; R0 <- R0 + 1
_5_ CMP  R0, 20
_6_ BRn  loop
```

(d) For this rescheduled code, how many cycles per loop iteration? What is the CPI?

Answer: [1 point] 6 cycles per iteration, .6 CPI

2. Pipelines, Superscalar, and Scheduling II.

Below is the same code as the previous question, but it has been loop unrolled once (two iterations converting into one):

```
loop:
    LD    R4,  R1,  0    ; R4 <- Mem[R1+0]
    LD    R5,  R2,  0    ; R5 <- Mem[R2+0]
    SUB   R6,  R4,  R5    ; R6 <- R4 - R5
    MUL   R7,  R6,  R6    ; R7 <- R6 * R6
    ADD   R3,  R3,  R7    ; R3 <- R3 + R7
    LD    R8,  R1,  1    ; R8 <- Mem[R1+1]
    LD    R9,  R2,  1    ; R9 <- Mem[R2+1]
    SUB   R10, R8,  R9    ; R10 <- R8 - R9
    MUL   R11, R10, R10    ; R11 <- R10 * R10
    ADD   R3,  R3,  R11    ; R3 <- R3 + R11
    ADD   R1,  R1,  2    ; R1 <- R1 + 2
    ADD   R2,  R2,  2    ; R2 <- R2 + 2
    ADD   R0,  R0,  2    ; R0 <- R0 + 2
    CMP   R0,  20
    BRn   loop
```

- (a) When executing the above code on a two-way (dual-issue) superscalar pipeline, how many cycles per loop iteration? What is the CPI?

Answer: [2 points] 13 cycles per iteration, .86 CPI

- (b) Assume for a moment you have an infinite-issue width processor (it can execute an unbounded number of independent instructions per cycle). What is the best cycles per loop and CPI possible?

Answer: [2 points] 7 cycles per iteration, .46 CPI

- (c) What is the minimum superscalar issue width necessary to achieve this best-case CPI?

Answer: [1 points] 3-wide

- (d) Show the rescheduled code that achieves the above best-base CPI. Instead of rewriting the code, just fill in the cycle the instruction begins execution in the “___” in the code below.

loop:

```
___ LD   R4,  R1,  0    ; R4 <- Mem[R1+0]
___ LD   R5,  R2,  0    ; R5 <- Mem[R2+0]
___ SUB  R6,  R4,  R5    ; R6 <- R4 - R5
___ MUL  R7,  R6,  R6    ; R7 <- R6 * R6
___ ADD  R3,  R3,  R7    ; R3 <- R3 + R7
___ LD   R8,  R1,  1    ; R8 <- Mem[R1+1]
___ LD   R9,  R2,  1    ; R9 <- Mem[R2+1]
___ SUB  R10, R8,  R9    ; R10 <- R8 - R9
___ MUL  R11, R10, R10   ; R11 <- R10 * R10
___ ADD  R3,  R3,  R11   ; R3 <- R3 + R11
___ ADD  R1,  R1,  2     ; R1 <- R1 + 2
___ ADD  R2,  R2,  2     ; R2 <- R2 + 2
___ ADD  R0,  R0,  2     ; R0 <- R0 + 2
___ CMP  R0,  20
___ BRn  loop
```

Answer: [5 points]

6

loop:

```
_1_ LD   R4, R1, 0    ; R4 <- Mem[R1+0]
_1_ LD   R5, R2, 0    ; R5 <- Mem[R2+0]
_3_ SUB  R6, R4, R5    ; R6 <- R4 - R5
_4_ MUL  R7, R6, R6    ; R7 <- R6 * R6
_5_ ADD  R3, R3, R7    ; R3 <- R3 + R7
_2_ LD   R8, R1, 1    ; R8 <- Mem[R1+1]
_2_ LD   R9, R2, 1    ; R9 <- Mem[R2+1]
_4_ SUB  R10, R8, R9   ; R10 <- R8 - R9
_5_ MUL  R11, R10, R10 ; R11 <- R10 * R10
_6_ ADD  R3, R3, R11   ; R3 <- R3 + R11
_1_ ADD  R1, R1, 2     ; R1 <- R1 + 2
_2_ ADD  R2, R2, 2     ; R2 <- R2 + 2
_3_ ADD  R0, R0, 2     ; R0 <- R0 + 2
_6_ CMP  R0, 20
_7_ BRn  loop
```

To help you understand the operation of caches, the following questions give the current state of the caches and memory and a single byte-sized memory operation (load or store). That is, all addresses are byte-granularity addresses.

Your task is to show us how the cache(s) and memory change when the instruction provided (`load` or `store`) is executed by crossing out the part that changed, and filling in what changed (for example, bring in the data block on a cache miss, set the tag, adjust the dirty bit, switch the LRU bit, write a dirty block back to memory, etc.).

For loads, also **circle the byte loaded by the instruction**. For stores, we supply both an address and a byte value to be written. Place that value into the cache, and **circle that byte as well**.

Each of the questions is independent (that is, what happened in the previous question has no impact on the next question).

All the caches are write-back, write-allocate caches. The block size in all cases is four bytes. As the size of the cache in the examples is larger than what we can fit on the paper, only the entries for the blocks in question are shown.

All of the addresses are given in *binary* (one bit per digit). To save space, all of the data values are given as hex values (4 bits per digit).

- For the single-core caches, a block can be in one of the following states:
 - C (*Clean*)
 - D (*Dirty*)
- For the set associative caches, if the *LRU* (least-recently used) field is either 0 (indicating that *Way 0* has the LRU block) or 1 (indicating that *Way 1* has the LRU block).
- For the multi-core cache examples, it uses an MSI protocol where each block in the cache can be:
 - I (*Invalid*)
 - S (*Shared*, which is read-only and clean)
 - M (*Modified*, which is read-write and dirty)

The memory also contains a state for each block:

- I (*Idle*, which indicates no cores are caching the block)
- S (*Shared*, which indicates at least one core is sharing the block and the memory is up to date)
- M (*Modified*, which indicates one and only one core is caching the block in the Modified state; the memory is not necessarily up to date)

In addition to the state, the memory has a *Sharers* field for tracking with processors (if any) are caching a particular block.

3. Direct Mapped Caches I [3 points]

```
load 1101010111010110
```

— Cache —						
Index	Tag	Data				State
		00	01	10	11	
⋮	⋮	⋮	⋮	⋮	⋮	⋮
0101110101	1101	BD	FE	[AB]	DE	D
⋮	⋮	⋮	⋮	⋮	⋮	⋮
1101011101	0101	CD	CA	CC	CB	C
⋮	⋮	⋮	⋮	⋮	⋮	⋮

— Memory —				
Address	Data			
	00	01	10	11
⋮	⋮	⋮	⋮	⋮
01010101110101xx	BB	BE	AE	FE
⋮	⋮	⋮	⋮	⋮
01011101011101xx	CD	CA	CC	CB
⋮	⋮	⋮	⋮	⋮
11010101110101xx	DE	DC	DD	DF
⋮	⋮	⋮	⋮	⋮
11011101011101xx	AD	BA	BE	CD
⋮	⋮	⋮	⋮	⋮

4. Direct Mapped Caches II [3 points]

```
store 0101110101110101 <- FF
```

— Cache —						
Index	Tag	Data				State
		00	01	10	11	
:	:	:	:	:	:	:
0101110101	1101	BD	FE	AB	DE	D
:	:	:	:	:	:	:
1101011101	0101	CD	FF FF	CC	CB	FF D
:	:	:	:	:	:	:

— Memory —				
Address	Data			
	00	01	10	11
:	:	:	:	:
01010101110101xx	BB	BE	AE	FE
:	:	:	:	:
01011101011101xx	CD	CA	CC	CB
:	:	:	:	:
11010101110101xx	DE	DC	DD	DF
:	:	:	:	:
11011101011101xx	AD	BA	BE	CD
:	:	:	:	:

5. Direct Mapped Caches III [3 points]

10

load 0101010111010110

— Cache —						
Index	Tag	Data				State
		00	01	10	11	
⋮	⋮	⋮	⋮	⋮	⋮	⋮
0101110101	1101 0101	BB BB	BE BE	AE [AE]	FE FE	C C
⋮	⋮	⋮	⋮	⋮	⋮	⋮
1101011101	0101	CD	CA	CC	CB	C
⋮	⋮	⋮	⋮	⋮	⋮	⋮

— Memory —				
Address	Data			
	00	01	10	11
⋮	⋮	⋮	⋮	⋮
01010101110101xx	BB	BE	[AE]	FE
⋮	⋮	⋮	⋮	⋮
01011101011101xx	CD	CA	CC	CB
⋮	⋮	⋮	⋮	⋮
11010101110101xx	BD BD	FE FE	AB AB	DE DE
⋮	⋮	⋮	⋮	⋮
11011101011101xx	AD	BA	BE	CD
⋮	⋮	⋮	⋮	⋮

6. Direct Mapped Caches IV [3 points]

11

```
store 1101110101110101 <- FF
```

— Cache —

Index	Tag	Data				State
		00	01	10	11	
:	:	:	:	:	:	:
0101110101	1101	BD	FE	AB	DE	D
:	:	:	:	:	:	:
1101011101	0101	AD AD	CA [FF]	BE BE	CD CD	D D
:	:	:	:	:	:	:

— Memory —

Address	Data			
	00	01	10	11
:	:	:	:	:
01010101110101xx	BB	BE	AE	FE
:	:	:	:	:
01011101011101xx	CD	CA	CC	CB
:	:	:	:	:
11010101110101xx	DE	DC	DD	DF
:	:	:	:	:
11011101011101xx	AD	BA	BE	CD
:	:	:	:	:

7. Set Associative Caches I [3 points]

12

load 1101110101010000

Way 0							LRU	Way 1					
Index	Tag	Data				State		Tag	Data				State
		00	01	10	11				00	01	10	11	
:	:	:	:	:	:	:		:	:	:	:	:	:
101010100	01011	AB	BD	EF	FA	C		1 0	11011	[BD]	BC	FE	DF
:	:	:	:	:	:	:	:	:	:	:	:	:	:

		— Memory —			
Address		Data			
		00	01	10	11
:	:	:	:	:	:
01011101010100xx	:	AB	BD	EF	FA
:	:	:	:	:	:
10101101010100xx	:	CC	DD	FB	AD
:	:	:	:	:	:
11011101010100xx	:	BF	CA	FE	FC
:	:	:	:	:	:

8. Set Associative Caches II [3 points]

13

load 1010110101010010

Index		Way 0					— Cache —		Way 1						
		Tag	Data						State	Tag	Data				State
			00	01	10	11					00	01	10	11	
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	
101010100	01011	AB	BD	EF	FA	C	1 0	11011 10101	00 CC	01 DD	10 [FB]	11 AD	0 C		
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮		

Address		Data			
		00	01	10	11
⋮	⋮	⋮	⋮	⋮	⋮
01011101010100xx	⋮	AB	BD	EF	FA
⋮	⋮	⋮	⋮	⋮	⋮
10101101010100xx	⋮	CC	DD	[FB]	AD
⋮	⋮	⋮	⋮	⋮	⋮
11011101010100xx	⋮	0/0 BD	0/0 BC	1/1 FE	0/0 DF
⋮	⋮	⋮	⋮	⋮	⋮

9. Multicore Caches I [3 points]

14

Processor 0: load 1110010101011010

— Processor 0's Cache —							— Processor 1's Cache —					
Index	Tag	Data				State	Tag	Data				State
		00	01	10	11			00	01	10	11	
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
0101010110	0101 1110	EF CE	EF ED	ED [DE]	EA EC	M S	1101	DE	AD	BE	EF	S
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
1101010110	0101	FA	BA	CE	AE	S	0101	FA	BA	CE	AE	S
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

— Memory —						
Address	Data				State	Sharers
	00	01	10	11		
⋮	⋮	⋮	⋮	⋮	⋮	⋮
01010101010110xx	ED AF	EA AB	ED EC	EF EA	M I	P0 —
⋮	⋮	⋮	⋮	⋮	⋮	⋮
01011101010110xx	FA	BA	CE	AE	S	P0, P1
⋮	⋮	⋮	⋮	⋮	⋮	⋮
11010101010110xx	DE	AD	BE	EF	S	P1
⋮	⋮	⋮	⋮	⋮	⋮	⋮
11100101010110xx	CE	ED	DE	EC	I S	# P0
⋮	⋮	⋮	⋮	⋮	⋮	⋮

10. Multicore Caches II [3 points]

15

Processor 1: load 0101010101011000

— Processor 0's Cache —							— Processor 1's Cache —						
Index	Tag	Data				State	Tag	Data				State	
		00	01	10	11			00	01	10	11		
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	
0101010110	0101	[AF]	AB	EC	EA	M S	1101 0101	00 [AF]	01 AB	10 EC	11 EA	S S	
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	
1101010110	0101	FA	BA	CE	AE	S	0101	FA	BA	CE	AE	S	
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	

— Memory —						
Address	Data				State	Sharers
	00	01	10	11		
⋮	⋮	⋮	⋮	⋮	⋮	⋮
01010101010110xx	00 [AF]	01 AB	10 EC	11 EA	M S	P0 P0, P1
⋮	⋮	⋮	⋮	⋮	⋮	⋮
01011101010110xx	FA	BA	CE	AE	S	P0, P1
⋮	⋮	⋮	⋮	⋮	⋮	⋮
11010101010110xx	DE	AD	BE	EF	S	P1
⋮	⋮	⋮	⋮	⋮	⋮	⋮
11100101010110xx	CE	ED	DE	EC	I	—
⋮	⋮	⋮	⋮	⋮	⋮	⋮

Note that some systems will idle 11010101010110xx and some will not. Both answers were accepted on the homework.

11. Multicore Caches III [3 points]

16

Processor 1: store 0101010101011001 <- FF

— Processor 0's Cache —							— Processor 1's Cache —						
Index	Tag	Data				State	Tag	Data				State	
		00	01	10	11			00	01	10	11		
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	
0101010110	0101	AF	AB	EC	EA	M I	1101 0101	DD AF	AD [FF]	BE EC	EE EA	S M	
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	
1101010110	0101	FA	BA	CE	AE	S	0101	FA	BA	CE	AE	S	
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	

— Memory —						
Address	Data				State	Sharers
	00	01	10	11		
⋮	⋮	⋮	⋮	⋮	⋮	⋮
01010101010110xx	DD	AA	DD	EE	M	P0 P1
⋮	⋮	⋮	⋮	⋮	⋮	⋮
01011101010110xx	FA	BA	CE	AE	S	P0, P1
⋮	⋮	⋮	⋮	⋮	⋮	⋮
11010101010110xx	DE	AD	BE	EF	S	P1
⋮	⋮	⋮	⋮	⋮	⋮	⋮
11100101010110xx	CE	ED	DE	EC	I	—
⋮	⋮	⋮	⋮	⋮	⋮	⋮

Note that some protocols update memory, and some do not. Since the data is going to be dirty immediately, it is cheaper (in terms of memory bandwidth and power) to do what is shown above. However, for the sake of completeness, some protocols choose to update memory at the same time. Both answers were accepted on the homework. In addition, some systems will idle 11010101010110xx and some will not. Both answers were accepted on the homework.

12. Multicore Caches IV [3 points]

17

Processor 0: store 1101010101011010 <- FF

— Processor 0's Cache —

Index	Tag	Data				State
		00	01	10	11	
⋮	⋮	⋮	⋮	⋮	⋮	⋮
0101010110	0101 1101	FF DE	FF AD	FF [FF]	FF EF	M I
⋮	⋮	⋮	⋮	⋮	⋮	⋮
1101010110	0101	FA	BA	CE	AE	S
⋮	⋮	⋮	⋮	⋮	⋮	⋮

— Processor 1's Cache —

Tag	Data				State
	00	01	10	11	
⋮	⋮	⋮	⋮	⋮	⋮
1101	DE	AD	BE	EF	S I
⋮	⋮	⋮	⋮	⋮	⋮
0101	FA	BA	CE	AE	S
⋮	⋮	⋮	⋮	⋮	⋮

— Memory —

Address	Data				State	Sharers
	00	01	10	11		
⋮	⋮	⋮	⋮	⋮	⋮	⋮
01010101010110xx	FF AF	FF AB	FF EC	FF EA	M I	P0 —
⋮	⋮	⋮	⋮	⋮	⋮	⋮
01011101010110xx	FA	BA	CE	AE	S	P0, P1
⋮	⋮	⋮	⋮	⋮	⋮	⋮
11010101010110xx	DE	AD	BE	EF	S M	P1 /P0, P1
⋮	⋮	⋮	⋮	⋮	⋮	⋮
11100101010110xx	CE	ED	DE	EC	I	—
⋮	⋮	⋮	⋮	⋮	⋮	⋮

13. Multicore Caches V [3 points]

18

Processor 0: store 0101110101011011 <- FF

— Processor 0's Cache —

Index	Tag	Data				State
		00	01	10	11	
⋮	⋮	⋮	⋮	⋮	⋮	⋮
0101010110	0101	AF	AB	EC	EA	M
⋮	⋮	⋮	⋮	⋮	⋮	⋮
1101010110	0101	FA	BA	CE	AA [FF]	S /M
⋮	⋮	⋮	⋮	⋮	⋮	⋮

— Processor 1's Cache —

Tag	Data				State
	00	01	10	11	
⋮	⋮	⋮	⋮	⋮	⋮
1101	DE	AD	BE	EF	S
⋮	⋮	⋮	⋮	⋮	⋮
0101	FA	BA	CE	AE	S I
⋮	⋮	⋮	⋮	⋮	⋮

— Memory —

Address	Data				State	Sharers
	00	01	10	11		
⋮	⋮	⋮	⋮	⋮	⋮	⋮
01010101010110xx	DD	AA	DD	EE	M	P0
⋮	⋮	⋮	⋮	⋮	⋮	⋮
01011101010110xx	FA	BA	CE	AE	S M	P0 / S / P0 P0
⋮	⋮	⋮	⋮	⋮	⋮	⋮
11010101010110xx	DE	AD	BE	EF	S	P1
⋮	⋮	⋮	⋮	⋮	⋮	⋮
11100101010110xx	CE	ED	DE	EC	I	—
⋮	⋮	⋮	⋮	⋮	⋮	⋮