Winston Shih
WXS190012
CS 2340.003

CS 2340 Assignment 5

Q1.

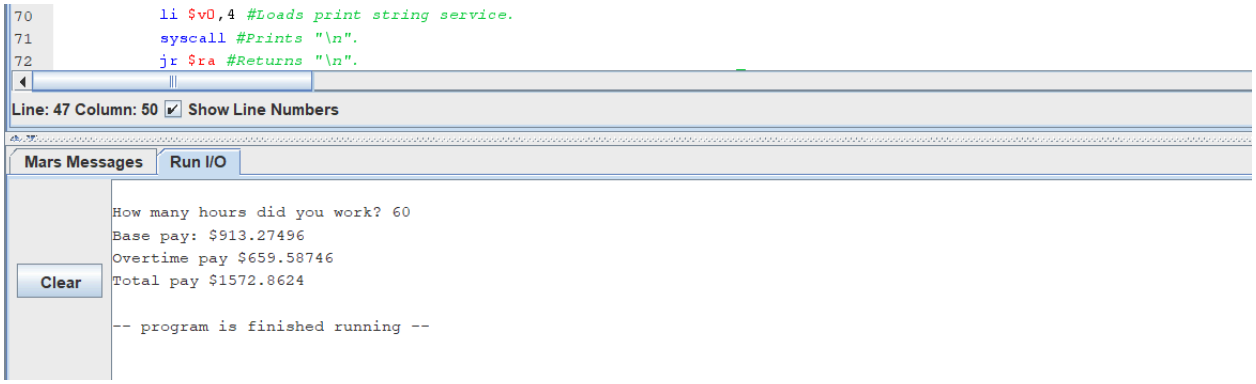
Edit Execute

WXS190012_Hw5_Q1.asm

```
#Winston Shih
#WXS190012
#CS 2340.003
.data #Data section of program.
     PAY_RATE: .float 22.55 #Represents value of PAY_RATE.
     BASE_HOURS: .float 40.50 #Represents value of BASE_HOURS.
     OT_MULTIPLIER: .float 1.5 #Represents value of OT_MULTIPLIER.
     noovertime: .float 0.0 #Represents value of overtime if hours<BASE_HOURS.
     hoursinput: .asciiz "How many hours did you work? " #Input prompt for hours of work.
     basepayoutput: .asciiz "Base pay: $" #Output prompt for base pay.
     overtimepayoutput: .asciiz "Overtime pay $"  #Output prompt got overtime pay.
     totalpayoutput: .asciiz "Total pay $" #Output prompt for total pay.
     newline: .asciiz "\n" #Represents endl.
.text #Executable portion of program.
     constants: l.s $f1,PAY_RATE #PAY_RATE=22.55
                l.s $f2,BASE_HOURS #BASE_HOURS=40.50
                l.s $f3,OT_MULTIPLIER #OT_MULTIPLIER=1.5
                l.s $f4,noovertime #overtime=0.0
     main: la $a0,hoursinput #Loads address for hoursinput prompt.
           li $v0,4 #Loads service for printing string.
           syscall #Prints "How many hours did you work? "
           li $v0,6 #Loads service to read input.
           syscall #Reads user's input for total hours they worked.
           mov.s $f12,$f0 #cin>>hours
           jal getBasePay #Calls getBasePay function with getBasePay(hours) call.
           mov.s $f5,$f0 #Result of getBasePay is stored in basePay.
           c.lt.s $f2, $f0 #Checks to see if BASE_HOURS<hours.
           bc1f print #If false, then jump to print.
           jal getOverTimePay #If true, then call getOverTimePay.
           mov.s $f4,$f0 #Moves return value of getOverTimePay function to overtime.
           add.s $f7,$f5,$f4 #Makes totalPay equal to basePay plus overtimePay.
     print: la $a0,basepayoutput #Loads address for basepayoutput prompt.
            li $v0,4 #Loads print string service.
            syscall #Prints "Base Pay : $".
            mov.s $f12,$f5 #Moves basePay's value to register $f12.
```

WXS190012_Hw5_Q1.asm

```
            li $v0,2 #Loads print float service.
            syscall #Prints base pay.
            jal endl #Prints a new line.
            la $a0,overtimepayoutput #Loads address of overtime pay prompt.
            li $v0,4 #Requests print string service.
            syscall #Prints "Overtime pay $".
            mov.s $f12,$f4 #Moves OT_MULTIPLIER value to register $f12.
            li $v0,2 #Loads print float service.
            syscall #Prints overtime pay.
            jal endl #Prints a new line.
            la $a0,totalpayoutput
            li $v0,4 #Loads print string service.
            syscall #Prints "Total pay $".
            mov.s $f12,$f7 #Moves value of totalPay to $f12.
            li $v0,2 #Loads print float service.
            syscall #Prints total pay.
            jal endl #Prints a new line.
     exit: li $v0,10 #Requests service to end program.
           syscall #Ends program.
     getBasePay: c.lt.s $f12,$f2 #Checks to see if BASE_HOURS < hoursWorked.
                 bc1t basepay #If previous condition is true, then jump to basepay.
                 mul.s $f0,$f2,$f1 #If BASE_HOURS>=hoursWorked, then basePay=hoursWorked*PAY_RATE.
                 jr $ra #Returns basePay.
     basepay: mul.s $f0,$f12,$f1 #basePay=BASE_HOURS*PAY_RATE
              jr $ra #Returns basePay.
     getOverTimePay: c.lt.s $f2,$f12 #Checks to see if BASE_HOURS<hoursWorked.
                     bc1t overtime #If hoursWorked is greater than BASE_HOURS, then jump to overtime function.
                     l.s $f0,noovertime #If hoursWorked<=BASE_HOURS, then overtimePay=0.0.
                     jr $ra #Returns overTimePay.
     overtime: sub.s $f12,$f12,$f2 #Stores result of (hoursWorked-BASE_HOURS) to $f12.
               mul.s $f6, $f1, $f3 #Stores result of  PAY_RATE*OT_MULTIPLIER to $f6.
               mul.s $f0,$f12,$f6 #Stores result of (hoursWorked-BASE_HOURS)*PAY_RATE*OT_MULTIPLIER to $f12.
               jr $ra #Returns overTimePay.
     endl: la $a0,newline #Loads address for newLine.
           li $v0,4 #Loads print string service.
```



Q2a.
Block size=2 words
Number of blocks in cache=4 blocks
Block offset bits=$\log_2 2$=1 bit
Index bits=$\log_2 4$=2 bits
46 and 47 both have tag of 00101 and index of 11, so hit
180 and 180 both have tag of 10110 and index of 10, so hit

Reference	Binary Address	Tag	Index (2 bits)	Offset (1 bit)	Hit/Miss
14	00001110	00001	11	0	Miss
172	10101100	10101	10	0	Miss
46	00101110	**00101**	**11**	0	Miss
176	10110000	10110	00	0	Miss
166	10100110	10100	11	0	Miss
6	00000110	00000	11	0	Miss
178	10110010	10110	01	0	Miss
42	00101010	00101	01	0	Miss
4	00000100	00000	10	0	Miss
180	10110100	**10110**	**10**	0	Miss
47	00101111	**00101**	**11**	1	Hit
180	10110100	**10110**	**10**	0	Hit

Q2b.
Block size=4 words
Number of blocks in cache=4 blocks
Offset bits=$\log_2 4$=2 bits
Index bits=$\log_2 4$=2 bits
46 and 47 have the same tag and index, so hit.

178 and 176 have the same tag and index, so hit.
4 and 6 have the same tag and index, so hit
180 and 180 have the same tag and index, so hit.

Reference	Binary Address	Tag	Index (2 bits)	Offset (1 bit)	Hit/Miss
14	00001110	0000	11	10	Miss
172	10101100	1010	11	00	Miss
46	00101110	**0010**	**11**	10	Miss
176	10110000	**1011**	**00**	00	Miss
166	10100110	1010	01	10	Miss
6	00000110	0000	01	10	Miss
178	10110010	**1011**	**00**	10	Hit
42	00101010	0010	10	10	Miss
4	00000100	0000	01	00	Hit
180	10110100	*1011*	*01*	00	Miss
47	00101111	**0010**	**11**	11	Hit
180	10110100	*1011*	*01*	00	Hit

Q2c.
Cache type=2-way
Block size=8 words
Word Offset=$\log_2 8$=3 bits
Number of blocks in cache=4 blocks
Number of sets=4/2=2
Set Offset bit=$\log_2 2$=1 bits
176 and 178 have same tag and set offset, so hit
46 and 42 have same tag and set offset, so hit
6 and 4 have same tag and set offset, so hit
42 and 47 have same tag and set offset, so hit
180 and 180 have same tag and set offset, so hit

Reference	Binary Address	Tag	Set Offset (1 bit)	Word Offset (3 bits)	Hit/Miss
14	00001110	0000	1	110	Miss
172	10101100	1010	1	100	Miss
46	00101110	**0010**	**1**	110	Miss

176	10110000	**1011**	**0**	000	Miss
166	10100110	1010	0	110	Miss
6	00000110	0000	0	110	Miss
178	10110010	**1011**	**0**	010	Hit
42	00101010	**0010**	**1**	010	Hit
4	00000100	0000	0	100	Hit
180	10110100	*1011*	*0*	100	Hit
47	00101111	**0010**	**1**	111	Hit
180	10110100	*1011*	*0*	100	Hit

Q2d.
Cache Type=4 way
Block size=4 words
Word Offset bit=$\log_2 4=2$
Number of blocks in cache=4 blocks
Set Offset bit=4 blocks/4=1 bit
46 and 47 have same tag and set offset, so hit
176 and 178 have same tag and set offset, so hit
6 and 4 have same tag and set offset, so hit
180 and 180 have same tag and set offset, so hit

Reference	Binary Address	Tag	Set Offset (1 bit)	Word Offset (2 bits)	Hit/miss
14	00001110	00001	1	10	Miss
172	10101100	10101	1	00	Miss
46	00101110	**00101**	**1**	10	Miss
176	10110000	**10110**	**0**	00	Miss
166	10100110	10100	1	10	Miss
6	00000110	00000	1	10	Miss
178	10110010	**10110**	**0**	10	Hit
42	00101010	00101	0	10	Miss
4	00000100	00000	1	00	Hit
180	10110100	*10110*	*1*	00	Miss

47	00101111	**<u>00101</u>**	**<u>1</u>**	11	Hit
180	10110100	*10110*	*1*	00	Hit

Q2e.
Cache 1 Miss Rate=Number of Misses/Total Memory References=10/12
Cache 1 Total Access Time=Miss Rate*Stall Time+access time=10/12*25 cycles+2 cycles=22.83 cycles
Cache 2 Miss Rate=Number of Misses/Total Memory References=8/12
Cache 2 Total Access Time=Miss Rate*Stall Time+access time=8/12*25 cycles+3 cycles=19.67 cycles
Cache 3 Miss Rate=Number of Misses/Total Memory References=6/12
Cache 3 Total Access Time=Miss Rate*Stall Time+access time=6/12*25 cycles+5 cycles=17.5 cycles
Cache 4 Miss Rate=Number of Misses/Total Memory References=8/12
Cache 4 Total Access Time=Miss Rate*Stall Time+access time=8/12*25 cycles+6 cycles=22.67 cycles
Since Cache 3 has the lowest total access time, Cache 3 has the best cache design.
Q2f.
Direct mapped cache with 32 word blocks
Cache data size=64 KiB*1024 byte/KiB*8 bits/byte=524288 bits
Word bits=32 words*32 bits/word=1024 bits
Number of blocks=524288 bits/1024 bits=512 blocks
Block offset=$\log_2$1024=10 bits
Index=$\log_2$512=9 bits
Tag bits=32-10-9=13 bits
Total Data Size=Cache Data Size=524288 bits
Total Index Size=Number of blocks*9 bits/block=512 blocks*9 bits/blocks=4608 bits
Total Tag Size=Number of blocks*13 bits/block=512 blocks*13 bits/blocks=6656 bits
Total Cache Size=Total Data Size+Total Index Size+Total Tag Size=524288 bits+4608 bits+6656 bits=535,552 bits=536 KiB
Direct mapped cache with 4 word blocks
Cache data size=524288 bits
Word bits=4 words*32 bits/word=128 bits
Number of blocks=524288 bits/128 bits=4096 blocks
Block offset=$\log_2$128=7 bits
Index=$\log_2$4096=12 bits
Tag bits=32-7-12=13 bits
Total cache size=524288 bits+4096 blocks*12 bits/block+4096 blocks*13 bits/block=551,936 bits=552 KiB
The first cache had a slower performance of 536 KiB compared to the second cache's 552 KiB performance because the larger data size can increase miss penalties (extra time needed to move data from main memory to cache when there is a cache miss). 32 word blocks would cause a direct mapped cache to have more cache misses than one with only 4 word blocks. Caches with 4-word blocks also have better spatial locality, which means addresses to certain

data locations are referenced faster and there are less cache misses compared to 32-word blocks.

Q3a.

12062, 10750, 10750, 2444, 44508, 35584, 4222, 2432,12060

Virtual Address (Decimal)	Virtual Address (Binary)	Virtual Page Number (Decimal)
12062	**0000 0000 0000 0000 001**0 1111 0001 1110	1
10750	**0000 0000 0000 0000 001**0 1001 1111 1110	1
10750	**0000 0000 0000 0000 001**0 1001 1111 1110	1
2444	**0000 0000 0000 0000 000**0 0100 1100 0110	0
44508	**0000 0000 0000 0000 101**0 1101 1101 1100	5
35584	**0000 0000 0000 0000 100**0 1011 0000 0000	4
4222	**0000 0000 0000 0000 000**1 0000 0111 1110	0
2432	**0000 0000 0000 0000 000**0 1001 1000 0000	0
12060	**0000 0000 0000 0000 001**0 1111 0001 1100	1

Virtual Address (Decimal)	TLB Hit/Miss	Page Table Hit/Miss	TLB Hit/Page Table hit/Page Fault
12062	TLB Miss	Page Table Miss	Page Fault
10750	TLB Hit	Page Table Hit	TLB Hit
10750	TLB Hit	Page Table Hit	TLB Hit
2444	TLB Miss	Page Table Miss	Page Fault
44508	TLB Miss	Page Table Hit	Page Table Hit
35584	TLB Miss	Page Table Hit	Page table Hit
4222	TLB Hit	Page Table Hit	TLB Hit
2432	TLB Hit	Page Table Hit	TLB Hit
12060	TLB Hit	Page Table Hit	TLB Hit

8KiB=8192 Bytes

$\log_2 8192 = 13$

$12062 = (10111100011110)_2$

$\text{Offset} = (0111100011110)_2$

Page Number=TLB Tag=$(1)_2$=1
Hit for TLB and Page Table
Largest page number=14
2*14=28
Tag 1->Physical Page number 28

Physical Page Number	Address
0011100	0111100011110

Valid	Tag	Physical page Number	LRU
1	20	10	1
1	2	5	2
1	3	6	3
1	1	28	0

10750=10100111111110
Offset=$(0100111111110)_2$
Page Number=TLB Tag=$(1)_2$=1
Tag 1->Page Number 28
Hit for TLB and Page Table

Physical Page Number	Address
0011100	0100111111110

Valid	Tag	Physical page Number	LRU
1	20	10	1
1	2	5	2
1	3	6	3
1	1	28	0

10750=$(10100111111110)_2$
Offset=$(0100111111110)_2$
Page Number=TLB Tag=$(1)_2$=1
Hit for TLB and Page table
Tag 1->Physical Page Number 28

Physical Page Number	Address

0011100	0100111111110

Valid	Tag	Physical page Number	LRU
1	20	10	1
1	2	5	2
1	3	6	3
1	1	28	0

$2444=(00100110001100)_2$
$Offset=(0100110001100)_2$
$Page\ Number=TLB\ Tag=0_2=0$
Miss for TLB and Page Table, so page fault
Largest page number=28
2*28=56
Tag 0->Physical Page number 56

Physical Page Number	Address
0111000	0100110001100

Valid	Tag	Physical page Number	LRU
1	20	10	2
1	2	5	3
1	0	56	0
1	1	28	1

$44508=(1010110111011100)_2$
$Offset=(0110111011100)_2$
$Page\ Number=TLB\ Tag=(101)_2=5$
Hit at Page Table, but miss at TLB table.
Tag 5->11

Physical Page Number	Address
0001011	0110111011100

Valid	Tag	Physical page	LRU

		Number	
1	20	10	3
1	5	11	0
1	0	56	1
1	1	28	2

35584=$(1000101100000000)_2$
Offset=$(0101100000000)_2$
Tag=$(100)_2$=4
TLB Miss, but hit at page table
Tag 4->Physical Page Number 9

Physical Page Number	Address
0001001	0101100000000

Valid	Tag	Physical page Number	LRU
1	4	9	0
1	5	11	1
1	0	56	2
1	1	28	3

4222=$(01000001111110)_2$
Offset=$(1000001111110)_2$
Tag=$(0)_2$=0
Hit for TLB and hit for page table
Tag 0->Physical Page Number 56

Physical Page Number	Address
0111000	1000001111110

Valid	Tag	Physical page Number	LRU
1	4	9	1
1	5	11	2

1	0	56	0
1	1	28	3

2432=$(00100110000000)_2$
Offset=$(0100110000000)_2$
Tag=$(0)_2$=0
Hit for TLB and page table
Tag 0->Page number 56

Physical Page Number	Address
0111000	0100110000000

Valid	Tag	Physical page Number	LRU
1	4	9	1
1	5	11	2
1	0	56	0
1	1	28	3

12060=$(10111100011100)_2$
Offset=$(0111100011100)_2$
Tag=$(1)_2$=1
Hit for page table and TLB
Tag 1->Physical Page Number 28

Physical Page Number	Address
0011100	0111100011100

Final state TLB table

Valid	Tag	Physical page Number	LRU
1	4	9	2
1	2	5	3
1	0	56	1
1	1	28	0

Final state page table

Valid	Physical Page Number

1	56
1	28
1	5
1	6
1	9
1	11
0	Disk
1	4
0	Disk
0	Disk
1	3
1	12
1	7
1	8
0	Disk
0	Disk
0	Disk
0	Disk
1	2
0	Disk
1	10
1	4

Q3b.

12062, 10750, 10750, 2444, 44508, 35584, 4222, 2432,12060

Virtual Address (Decimal)	Virtual Address (Binary)	Virtual Page Number (Decimal)
12062	**0000 0000 0000 0000 0010** 1111 0001 1110	2
10750	**0000 0000 0000 0000 0010** 1001 1111 1110	2

10750	**0000 0000 0000 0000 0010** 1001 1111 1110	2
2444	**0000 0000 0000 0000 0000** 0100 1100 0110	0
44508	**0000 0000 0000 0000 1010** 1101 1101 1100	10
35584	**0000 0000 0000 0000 1000** 1011 0000 0000	8
4222	**0000 0000 0000 0000 0001** 0000 0111 1110	1
2432	**0000 0000 0000 0000 0000** 1001 1000 0000	0
12060	**0000 0000 0000 0000 0010** 1111 0001 1100	2

4KiB=4096 Bytes
$\log_2 4096=12$
$12062=(10111100011110)_2$
$\text{Offset}=(111100011110)_2$
$\text{Page Number=TLB Tag}=(10)_2=2$
Hit for Page Table and Hit for TLB table
Tag 2->Page 5

Physical Page Number	Address
0000101	111100011110

Valid	Tag	Physical page number	LRU
1	20	10	1
1	2	5	0
1	3	6	2
0	0	15	3

$10750=(10100111111110)_2$
$\text{Offset}=(100111111110)_2$
$\text{Page Number=TLB Tag}=(10)_2=2$
Hit for Page Table and Hit for TLB table

Physical Page Number	Address
0000101	100111111110

Valid	Tag	Physical Page Number	LRU

1	20	10	1
1	2	5	0
1	3	6	2
0	0	15	3

10750=10100111111110
Offset=$(100111111110)_2$
Page Number=TLB Tag=$(10)_2$=2
Hit for Page Table and Hit for TLB table
Tag 2->Physical Page Number 5

Physical Page Number	Address
0000101	100111111110

Valid	Tag	Physical Page Number	LRU
1	20	10	1
1	2	5	0
1	3	6	2
0	0	15	3

2444=$(00100110001100)_2$
Offset=$(100110001100)_2$
Page Number=TLB Tag=$(00)_2$=0
Miss for TLB and Page Table
Largest page number=14
14*2=28
Tag 0->Physical Page Number 28

Physical Page Number	Address
0011100	100110001100

Valid	Tag	Physical Page Number	LRU
1	20	10	2
1	2	5	1
1	3	6	3

1	0	28	0

44508=$(1010110111011100)_2$
Offset=$(110111011100)_2$
Page Number=TLB Tag=$(1010)_2$=10
Hit at Page Table, but miss at TLB table.
Tag 10->Physical Page Number 3

Physical Page Number	Address
0000011	110111011100

Valid	Tag	Physical Page Number	LRU
1	20	10	3
1	2	5	2
1	10	3	0
1	0	28	1

35584=$(1000101100000000)_2$
Offset=$(101100000000)_2$
Tag=$(1000)_2$=8
TLB Miss and page table miss, so page fault
Largest page number=28
28*2=56
Tag 8->Physical Page Number 56

Physical Page Number	Address
0111000	101100000000

Valid	Tag	Physical Page Number	LRU
1	8	56	0
1	2	5	3
1	10	3	1
1	0	28	2

4222=$(01000001111110)_2$
Offset=$(000001111110)_2$
Tag=$(1)_2$=1

Miss for page table and miss for TLB, so Page Fault.
Largest page number=56
56*2=112
Tag 1->Physical Page Number 112

Physical Page Number	Address
1111010	000001111110

Valid	Tag	Physical Page Number	LRU
1	8	56	1
1	1	112	0
1	10	3	2
1	0	28	3

2432=$(00100110000000)_2$
Offset=$(100110000000)_2$
Tag=(00)2=0
Hit for TLB and hit for page tale
Tag 0->Physical Page Number 28

Physical Page Number	Address
0011100	100110000000

Valid	Tag	Physical Page Number	LRU
1	8	56	2
1	1	112	1
1	10	3	3
1	0	28	0

12060=$(10111100011100)_2$
Offset=$(111100011100)_2$
Tag=$(10)_2$=2
Hit for TLB and Page Table

Virtual Address (Decimal)	TLB Hit/Miss	Page Table Hit/Miss	TLB Hit/Page Table hit/Page Fault
12062	TLB Hit	Page Table Hit	TLB Hit

10750	TLB Hit	Page Table Hit	TLB Hit
10750	TLB Hit	Page Table Hit	TLB Hit
2444	TLB Miss	Page Table Miss	Page Fault
44508	TLB Miss	Page Table Hit	Page Table Hit
35584	TLB Miss	Page Table Miss	Page Fault
4222	TLB Miss	Page Table Miss	Page Fault
2432	TLB Hit	Page Table Hit	TLB Hit
12060	TLB Hit	Page Table Hit	TLB Hit

Final TLB table state

Valid	Tag	Physical Page Number	LRU
1	8	56	2
1	1	112	1
1	3	6	3
1	0	28	0

Final Page Table

Valid	Physical Page Number
1	28
1	112
1	5
1	6
1	9
1	11
0	Disk
1	4
1	56
0	Disk
1	3

1	12
1	7
1	8
0	Disk
0	Disk
0	Disk
0	Disk
1	2
0	Disk
1	10
1	14

Some advantages of smaller pages are that these pages reduce fragmentation of data (data that is split up into unnecessary secondary caches of memory) and have better temporal and spatial locality. A better temporal (if data location is referenced, it will be referenced again) and spatial locality (if data address is referenced, then its nearby addresses will be referenced again) means less cache misses and smaller miss penalties, which improves memory performance of cache. Reduced fragmentation of data (internal or external) also helps improve the cache's performance by ensuring less data is being used inefficiently. The disadvantages of smaller pages are that it has a bigger page table, has more page faults, and increased overhead for reading or writing pages. Smaller pages cause more overhead by needing more page table entries. The overhead can cause memory access time to be longer compared to access time of bigger pages. More page faults mean that there is a higher likelihood of system crashes, process dumps, and reduced application performance.

Q4.

This MIPs subset is an implementation of a particular subset of MIPs instructions. This instruction subset includes instructions like memory-reference, arithmetic-logical, and branch equal or jump instructions. The first instructions are memory-reference, which include load word and store word. The next instructions are arithmetic-logical instructions, which include add, sub, and, or, slt. The last instructions are branch equal and jump instructions. In this MIPs subset figure, it shows that the MIPs architecture has 4 main components: instruction memory, registers, arithmetic logic unit, and control unit. The model also shows the MIPs architecture's control lines and multiplexors.

The instruction memory is the part of the MIPs subset that stores the instructions for the CPU processor to execute. The default block size for each instruction is a 32-bit word. The registers are storage locations in the MIPs subset that the CPU uses to temporarily store the data and addresses from the instructions in the source code. There are usually 32 registers in a MIPs subset and each register has a 32 bit word cache. The ALU does all the MIPs arithmetic

or logic operations that are specified in the code. These operations include addition, subtraction, multiplication, division, and bitwise operations (like bitwise AND, bitwise OR, bitwise left, bitwise right, bitwise NOT, or bitwise exclusive OR). Memory-reference, arithmetic-logical, and branv\ch instructions all use the ALU after they are read in registers. The memory-reference instructions use the ALU to calculate address for instructions, while arithmetic-logical instructions use ALU to execute operations. Branches use ALU for their comparison operations. After using ALU, memory-reference instructions will access memory to perform read or write operations. Arithmetic-logical or load instructions will perform write operation on data from ALU or its memory back to the registers. After ALU is utilized, the branch instruction will need to update the instruction address for the next comparison operation. If it can not achieve that, it will have to increment the address by four to get the new address for next instruction. The control unit manipulates flow of data and decides what data operation should be performed. After receiving its opcode, it determines the appropriate value to set control lines to perform the right operation in the processor.

The purpose of the multiplexor (data selector) in the MIPs subset is to determine the appropriate data sources for ALU and register files to accept. After making its input selections, the multiplexor combines those inputs into a data stream to optimize the subset's efficiency. In this figure, there are three multiplexors. The multiplexor on the top of the figure is used to select the address of the next instruction after the current instruction is completed. The multiplexor in the middle of the model selects operands that will be used by ALU (registers, immediate values, or PC). The bottom multiplexor of this MIPs implementation will select data from an ALU operation, register value, or immediate value that will be sent to the destination register or processor's memory by determining if the data came from registers or the instruction's offset field. The control lines in the subset are used by the control unit to regulate the access and use of data or addresses lines in the multiplexors, instruction memory, arithmetic logic unit, or any other component of the MIPs subset.

The processor of the MIPs subset executes an instruction first by fetching instructions from its instruction memory by sending its PC (program counter) to its memory. After reading the instruction, it will read one register or two registers. If the instruction says load word, then it will only read one instruction. Otherwise, the instruction will usually tell the processor to read two registers. After these first two steps, the remaining actions that the processor will execute will differ based on what instruction classes are in code. For memory-reference, arithmetic-logical, and branch instructions, the processor will execute the same actions that are independent from the exact instructions it is based on. One of the two adders is responsible for writing the value of PC. Before the processor can implement the instruction, it needs its control unit to decode the instructions to know what data sources to use as input and what arithmetic and logic operations to run. After the instructions are fully decoded, the multiplexors are responsible for selecting the right data sources for the processor's Arithmetic Logic Unit and register file. The top multiplexor determines what is the next value for the branch destination address. An "AND" bitwise branch instruction gate (combines zero address output and control signal) controls what the multiplexor does. The middle multiplexer will move the ALU or data memory's output into a register file that it will return as its output. The ALU and register file receive the data sources, which causes the ALU to perform the requested operations from the instructions. The bottom multiplexor will take in the input from the second ALU in the subset and determine if it came from registers or an

instruction’s offset field. If the input is a load or store instruction, then the multiplexor will send input to the processor's memory. If the instruction is from a branch or arithmetic-logical operation, then the multiplexor will send input to the destination register. After all the ALU operations are completed, its results are sent to a destination register (register file or the subset memory). The control lines decide the operations ALU executes. These operations include whether data memory reads, data memory writes, or register writes. The data memory writes on a load instruction and reads on a store operation. A load or arithmetic-logical instruction is needed for a register file to be written in.

Q5. A data hazard is a hazard that happens when an instruction in code needs to reference the result of a previous instruction in the pipeline, but the CPU has not computed the result yet. An example of a data hazard is a Read-After-Write data hazard, which is when a data pipeline causes an instruction to potentially read an operand before a previous instruction can write to the operand. A structural hazard is a hazard when a computer hazard is unable to support certain instruction combinations since the specific combination of instructions need to use the same CPU part or resource. An example of a structural hazard is when there is only one Arithmetic Logic Unit available to execute a combination of instructions in the source code.