Department of Computer Science
National Tsing Hua University
EECS403000 Computer Architecture

Spring 2024, Homework 2

Due date: **April 11, 2024 23:59 pm**

To inspect the Assembly code of the program, follow the steps below.

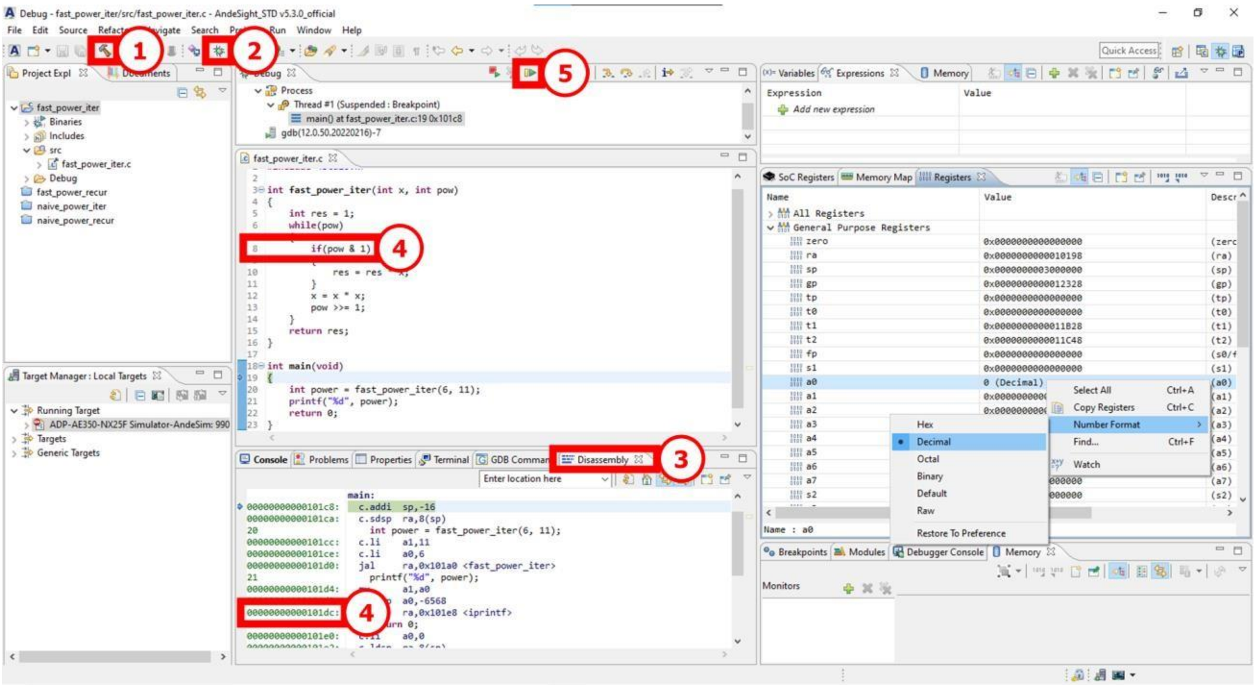
(1) Build the program with the Debug configuration.

(2) Start debugging the desired program as an Application Program.

(3) Navigate to the Disassembly view to examine the generated assembly code.

(4) To insert breakpoints, double-click on the Assembly code or the corresponding C code lines on the left.

(5) Press Resume in the debug window to proceed to the next breakpoint.

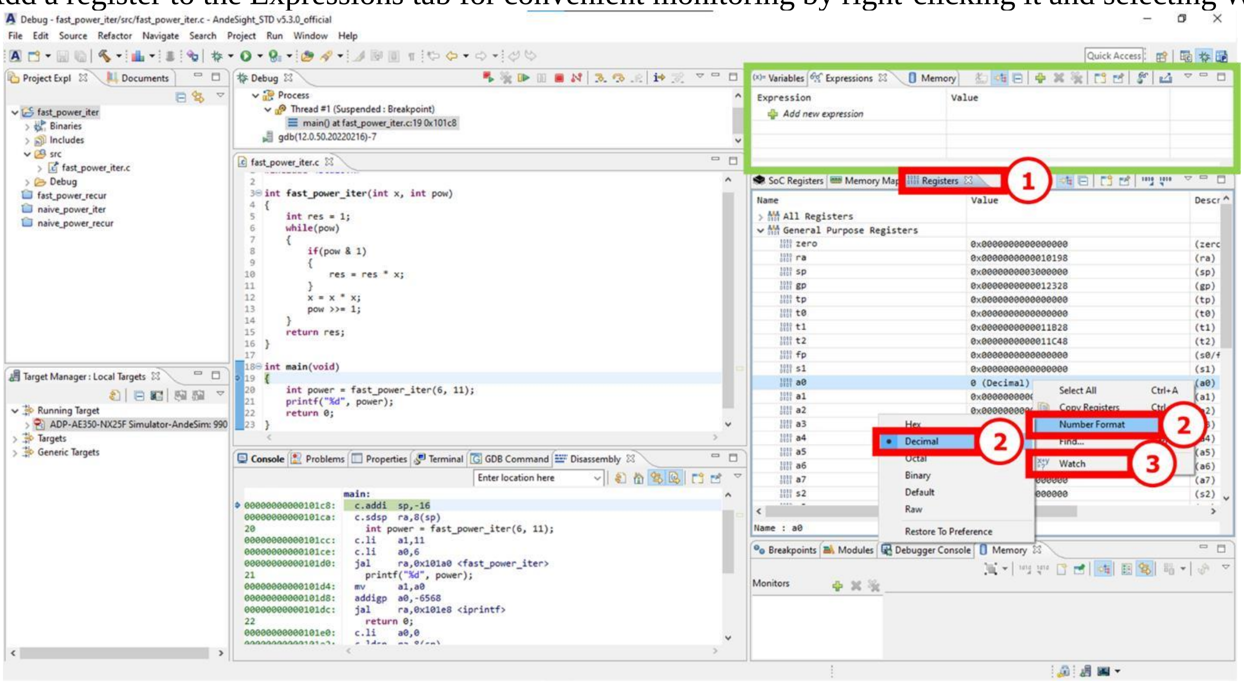


To observe register value changes, Start Debugging as an Application Program and follow the steps:

(1) Access the Registers tab and expand the General Purpose Registers section to view their current values.

(2) Customize the Number format by right-clicking a register and choosing the desired format.

(3) Add a register to the Expressions tab for convenient monitoring by right-clicking it and selecting Watch.



1. (35 points)

This question explores the fast power algorithm implemented in two ways (iterative and recursive) using AndeSight™ with the setup similar to Homework 1. We will analyze the source code for **`fast_power_iter.c`** and **`fast_power_recur.c`**. The default optimization level is -Og by default unless stated otherwise. **For each question, attach screenshots of AndeSight™ to support your answer. No points will be given if the screenshot is missing**. **Hint**: Use the “Debug” button in the toolbar and carefully insert breakpoints in between instructions. Press the “Resume” button in the debug window to move to the next breakpoint. You can also refer to the RISC-V Specs document if you encounter some difficulty understanding the Assembly code generated by AndeSight™.

(a) (5 points) ***RISC-V Calling Convention***

Compilers typically translate functions into subroutine and perform function calls using a jump instruction following a calling convention. While function calls can be inefficient, they are essential in programming. Locate the starting and ending memory addresses of the code memory allocated to the **`fast_power_recur`** and **`fast_power_iter`** subroutines by examining the Assembly code. Identify how these subroutines are called within main and write them down in the Reference field in the table.

Subroutine	Starting memory address	Ending memory address	Reference
`fast_power_recur`	00000000000104a8	00000000000104e0	**`jal ra,0x104a8 <fast_power_recur>`**
`fast_power_iter`	00000000000104a8	00000000000104c4	**`jal ra,0x104a8 <fast_power_iter>`**

```
                   main:
00000000000104e4:    c.addi sp,-16
00000000000104e6:    c.sdsp ra,8(sp)
20                     int power = fast_power_recur(6, 11);
00000000000104e8:    c.li a1,11
00000000000104ea:    c.li a0,6
00000000000104ec:    jal ra,0x104a8 <fast_power_recur>
21                     printf("%d", power);
00000000000104f0:    c.mv a1,a0
00000000000104f2:    addi a0,gp,-8
00000000000104f6:    jal ra,0x105b0 <printf>
23                 }
00000000000104fa:    c.li a0,0
00000000000104fc:    c.ldsp ra,8(sp)
00000000000104fe:    c.addi sp,16
0000000000010500:    c.jr ra
0000000000010502:    unimp
```

```
5                      if(pow == 0)
                   fast_power_recur:
00000000000104a8:    c.bnez a1,0x104b0 <fast_power_recur+8>
7                            return 1;
00000000000104aa:    c.li a0,1
16                 }
00000000000104ac:    ret
4                  {
00000000000104b0:    c.addi sp,-32
00000000000104b2:    c.sdsp ra,24(sp)
00000000000104b4:    c.sdsp s0,16(sp)
00000000000104b6:    c.sdsp s1,8(sp)
00000000000104b8:    c.mv s0,a1
00000000000104ba:    c.mv s1,a0
10                     int m = fast_power_recur(x, pow >> 1);
00000000000104bc:    sraiw a1,a1,0x1
00000000000104c0:    jal ra,0x104a8 <fast_power_recur>
11                     if(pow & 1)
00000000000104c4:    bbs s0,0,0x104d8 <fast_power_recur+48>
15                     return m * m;
00000000000104c8:    mulw a0,a0,a0
16                 }
00000000000104cc:    c.ldsp ra,24(sp)
00000000000104ce:    c.ldsp s0,16(sp)
00000000000104d0:    c.ldsp s1,8(sp)
00000000000104d2:    c.addi16sp sp,32
00000000000104d4:    ret
13                           return x * m * m;
00000000000104d8:    mulw s1,s1,a0
00000000000104dc:    mulw a0,s1,a0
00000000000104e0:    j 0x104cc <fast_power_recur+36>
19                 {
```

```
                   main:
00000000000104c8:    c.addi sp,-16
00000000000104ca:    c.sdsp ra,8(sp)
20                     int power = fast_power_iter(6, 11);
00000000000104cc:    c.li a1,11
00000000000104ce:    c.li a0,6
00000000000104d0:    jal ra,0x104a8 <fast_power_iter>
21                     printf("%d", power);
00000000000104d4:    c.mv a1,a0
00000000000104d6:    addi a0,gp,-8
00000000000104da:    jal ra,0x10592 <printf>
23                 }
00000000000104de:    c.li a0,0
00000000000104e0:    c.ldsp ra,8(sp)
00000000000104e2:    c.addi sp,16
00000000000104e4:    c.jr ra
```

```
                   fast_power_iter:
00000000000104a8:    c.mv a5,a0
5                      int res = 1;
00000000000104aa:    c.li a0,1
6                      while(pow)
00000000000104ac:    j 0x104b8 <fast_power_iter+16>
12                           x = x * x;
00000000000104b0:    mulw a5,a5,a5
13                           pow >>= 1;
00000000000104b4:    sraiw a1,a1,0x1
6                      while(pow)
00000000000104b8:    c.beqz a1,0x104c4 <fast_power_iter+28>
8                            if(pow & 1)
00000000000104ba:    bbc a1,0,0x104b0 <fast_power_iter+8>
10                               res = res * x;
00000000000104be:    mulw a0,a5,a0
00000000000104c2:    c.j 0x104b0 <fast_power_iter+8>
16                 }
00000000000104c4:    ret
19                 {
```

(b) (10 points) ***RISC-V Calling Convention***

The RISC-V calling convention requires the callee to preserve the values of specific registers across function calls. Examine the Assembly code for **`fast_power_recur`** function. Find and record the instructions (and the memory locations) that save these registers. Extend the table below to show how these instructions affect the stack. Include the saved register names and corresponding stack offsets. Order the table by the increasing order of stack offset.

Code memory address	Instruction	Saved register	Stack offset
`00000000000104b6`	**`c.sdsp s1,8(sp)`**	**`s1`**	**`8`**
`00000000000104b4`	**`c.sdsp s0,16(sp)`**	**`s0`**	**`16`**
`00000000000104b2`	**`c.sdsp ra,24(sp)`**	**`ra`**	**`24`**

```
00000000000104a6:   c.jr ra
 5                    if(pow == 0)
                  fast_power_recur:
00000000000104a8:   c.bnez a1,0x104b0 <fast_power_recur+8>
 7                        return 1;
00000000000104aa:   c.li a0,1
16                }
00000000000104ac:   ret
 4                {
00000000000104b0:   c.addi sp,-32
00000000000104b2:   c.sdsp ra,24(sp)
00000000000104b4:   c.sdsp s0,16(sp)
00000000000104b6:   c.sdsp s1,8(sp)
00000000000104b8:   c.mv s0,a1
00000000000104ba:   c.mv s1,a0
10                    int m = fast_power_recur(x, pow >> 1);
00000000000104bc:   sraiw a1,a1,0x1
00000000000104c0:   jal ra,0x104a8 <fast_power_recur>
11                    if(pow & 1)
00000000000104c4:   bbs s0,0,0x104d8 <fast_power_recur+48>
15                    return m * m;
00000000000104c8:   mulw a0,a0,a0
16                }
00000000000104cc:   c.ldsp ra,24(sp)
00000000000104ce:   c.ldsp s0,16(sp)
00000000000104d0:   c.ldsp s1,8(sp)
00000000000104d2:   c.addi16sp sp,32
00000000000104d4:   ret
13                        return x * m * m;
00000000000104d8:   mulw s1,s1,a0
00000000000104dc:   mulw a0,s1,a0
00000000000104e0:   j 0x104cc <fast_power_recur+36>
19                {
```

(c) (10 points) ***Effects of the compiler on RISC-V Assembly Code***

To make common cases fast, compilers can allocate application program variable to processor registers, which are much faster than memory. Compile both **`fast_power_iter`** and **`fast_power_recur`** functions with -O0 optimization flag and answer the following questions:

(i) Just before jumping to the corresponding subroutine from **`main`**, which registers hold the parameters **`x`** and **`pow`**, and what values do they contain?

(ii) Immediately after returning from the subroutine (i.e. just before the next instruction after the jump), which register stores the return value, and what is its value?

Function	Parameter **`x`**		Parameter **`pow`**		Return value	
	Register	Value	Register	Value	Register	Value
`fast_power_iter`	**`a0`**	**`6`**	**`a1`**	**`11`**	**`a0`**	**`362797056`**
`fast_power_recur`	**`a0`**	**`6`**	**`a1`**	**`11`**	**`a0`**	**`362797056`**

```
main:
0000000000010514:   c.addi sp,-32
0000000000010516:   c.sdsp ra,24(sp)
0000000000010518:   c.sdsp s0,16(sp)
000000000001051a:   c.addi4spn s0,sp,32
20                    int power = fast_power_iter(6, 11);
000000000001051c:   c.li a1,11
000000000001051e:   c.li a0,6
0000000000010520:   jal ra,0x104a8 <fast_power_iter>
0000000000010524:   c.mv a5,a0
0000000000010526:   sw a5,-20(s0)
21                    printf("%d", power);
000000000001052a:   lw a5,-20(s0)
000000000001052e:   c.mv a1,a5
0000000000010530:   addi a0,gp,-8
0000000000010534:   jal ra,0x105f0 <printf>
22                    return 0;
0000000000010538:   c.li a5,0
23                  }
000000000001053a:   c.mv a0,a5
000000000001053c:   c.ldsp ra,24(sp)
000000000001053e:   c.ldsp s0,16(sp)
0000000000010540:   c.addi16sp sp,32
0000000000010542:   c.jr ra
```

```
main:
0000000000010530:   c.addi sp,-32
0000000000010532:   c.sdsp ra,24(sp)
0000000000010534:   c.sdsp s0,16(sp)
0000000000010536:   c.addi4spn s0,sp,32
20                    int power = fast_power_recur(6, 11);
0000000000010538:   c.li a1,11
000000000001053a:   c.li a0,6
000000000001053c:   jal ra,0x104a8 <fast_power_recur>
0000000000010540:   c.mv a5,a0
0000000000010542:   sw a5,-20(s0)
21                    printf("%d", power);
0000000000010546:   lw a5,-20(s0)
000000000001054a:   c.mv a1,a5
000000000001054c:   addi a0,gp,-8
0000000000010550:   jal ra,0x1060c <printf>
22                    return 0;
0000000000010554:   c.li a5,0
23                  }
0000000000010556:   c.mv a0,a5
0000000000010558:   c.ldsp ra,24(sp)
000000000001055a:   c.ldsp s0,16(sp)
000000000001055c:   c.addi16sp sp,32
000000000001055e:   c.jr ra
```

```
fast_power_iter:
00000000000104a8:   c.addi16sp sp,-48
00000000000104aa:   c.sdsp s0,40(sp)
00000000000104ac:   c.addi4spn s0,sp,48
00000000000104ae:   c.mv a5,a0
00000000000104b0:   c.mv a4,a1
00000000000104b2:   sw a5,-36(s0)
00000000000104b6:   c.mv a5,a4
00000000000104b8:   sw a5,-40(s0)
 5                    int res = 1;
00000000000104bc:   c.li a5,1
00000000000104be:   sw a5,-20(s0)
 6                    while(pow)
00000000000104c2:   c.j 0x104fc <fast_power_iter+84>
 8                        if(pow & 1)
00000000000104c4:   lw a5,-40(s0)
00000000000104c8:   c.andi a5,1
00000000000104ca:   bfos a5,a5,31,0
00000000000104ce:   c.beqz a5,0x104e0 <fast_power_iter+56>
10                            res = res * x;
00000000000104d0:   lw a4,-20(s0)
00000000000104d4:   lw a5,-36(s0)
00000000000104d8:   mulw a5,a4,a5
00000000000104dc:   sw a5,-20(s0)
12                        x = x * x;
00000000000104e0:   lw a4,-36(s0)
00000000000104e4:   lw a5,-36(s0)
00000000000104e8:   mulw a5,a4,a5
00000000000104ec:   sw a5,-36(s0)
13                        pow >>= 1;
00000000000104f0:   lw a5,-40(s0)
00000000000104f4:   sraiw a5,a5,0x1
00000000000104f8:   sw a5,-40(s0)
 6                    while(pow)
00000000000104fc:   lw a5,-40(s0)
0000000000010500:   bfos a5,a5,31,0
0000000000010504:   c.bnez a5,0x104c4 <fast_power_iter+28>
15                    return res;
0000000000010506:   lw a5,-20(s0)
16                  }
000000000001050a:   c.mv a0,a5
000000000001050c:   c.ldsp s0,40(sp)
000000000001050e:   c.addi16sp sp,48
0000000000010510:   ret
```

fp	0x0000000000000000
s1	0x0000000000000000
a0	362797056 (Decimal)
a1	0x0000000000000000

```
fast_power_recur:
00000000000104a8:   c.addi16sp sp,-48
00000000000104aa:   c.sdsp ra,40(sp)
00000000000104ac:   c.sdsp s0,32(sp)
00000000000104ae:   c.addi4spn s0,sp,48
00000000000104b0:   c.mv a5,a0
00000000000104b2:   c.mv a4,a1
00000000000104b4:   sw a5,-36(s0)
00000000000104b8:   c.mv a5,a4
00000000000104ba:   sw a5,-40(s0)
 5                    if(pow == 0)
00000000000104be:   lw a5,-40(s0)
00000000000104c2:   bfos a5,a5,31,0
00000000000104c6:   c.bnez a5,0x104cc <fast_power_recur+36>
 7                        return 1;
00000000000104c8:   c.li a5,1
00000000000104ca:   c.j 0x10524 <fast_power_recur+124>
10                    int m = fast_power_recur(x, pow >> 1);
00000000000104cc:   lw a5,-40(s0)
00000000000104d0:   sraiw a5,a5,0x1
00000000000104d4:   bfos a4,a5,31,0
00000000000104d8:   lw a5,-36(s0)
00000000000104dc:   c.mv a1,a4
00000000000104de:   c.mv a0,a5
00000000000104e0:   jal ra,0x104a8 <fast_power_recur>
00000000000104e4:   c.mv a5,a0
00000000000104e6:   sw a5,-20(s0)
11                    if(pow & 1)
00000000000104ea:   lw a5,-40(s0)
00000000000104ee:   c.andi a5,1
00000000000104f0:   bfos a5,a5,31,0
00000000000104f4:   c.beqz a5,0x10514 <fast_power_recur+108>
13                        return x * m * m;
00000000000104f6:   lw a4,-36(s0)
00000000000104fa:   lw a5,-20(s0)
00000000000104fe:   mulw a5,a4,a5
0000000000010502:   bfos a5,a5,31,0
0000000000010506:   lw a4,-20(s0)
000000000001050a:   mulw a5,a4,a5
000000000001050e:   bfos a5,a5,31,0
0000000000010512:   c.j 0x10524 <fast_power_recur+124>
15                    return m * m;
0000000000010514:   lw a4,-20(s0)
0000000000010518:   lw a5,-20(s0)
000000000001051c:   mulw a5,a4,a5
0000000000010520:   bfos a5,a5,31,0
16                  }
0000000000010524:   c.mv a0,a5
0000000000010526:   c.ldsp ra,40(sp)
0000000000010528:   c.ldsp s0,32(sp)
000000000001052a:   c.addi16sp sp,48
000000000001052c:   ret
19                  {
```

s1	0x0000000000000000
a0	362797056 (Decimal)
a1	0x0000000000000000
a2	0x0000000000000000

(d) (10 points) ***Effects of great ideas on performance***

Pipeline execution and parallelization are two key techniques for enhancing performance. Consider **`fast_power_iter.c`** and disregard overheads from parallelization and data transmission. Here's an approach for pipelined execution and parallelization using three cores and a two-stage pipeline. Assume that each core fetches and executes the instructions sequentially.

Core A (Pipeline Stage 1): Calculates **`x`** ← **`x * x`**, and passes the new **`x`** to Core C.

Core B (Pipeline Stage 1): Calculates **`pow`** ← **`pow >> 1`**, and passes the new **`pow`** to Core C.

Core C (Pipeline Stage 2): Checks if pow is zero. If true, execute a jump and there is nothing left to do. Otherwise, it calculates **`res`** ← **`res * x`** if **`pow`** is odd, where **`res`** and **`pow`** are results from Core A and Core B in the previous cycle.

Time axis →

Core A & B	Core C						
	Core A & B	Core C					
		Core A & B	Core C				
			Core A & B	Core C			
				Core A & B	Core C		
					Core A & B	Core C	
						Core A & B	…
							…

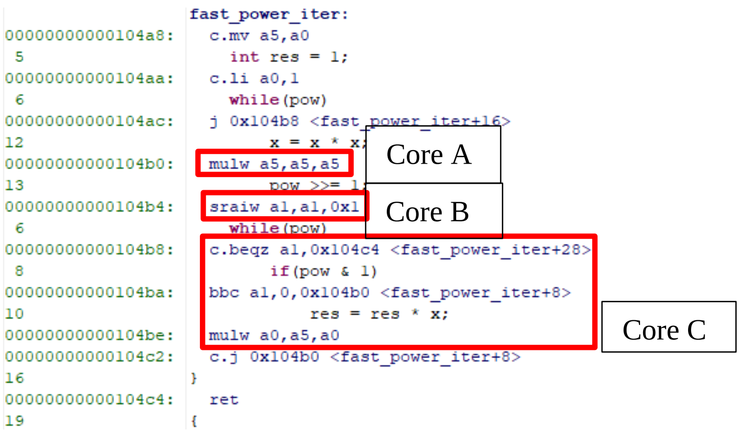
In other words, Cores A and B operate in parallel, while Core C processes their results in a pipeline fashion. Analyze the code of **`fast_power_iter`** function in **`fast_power_iter.c`**, identify the instructions for each core, and record their cycle counts. Explain whether achieving a speedup of 2 for the pipelined parallelized function is possible.

Core C: 6	**Core A: 4** **Core B: 1**			
	Core C: 3	**Core A: 1** **Core B: 1**		
		Core C: 2	**Core A: 4** **Core B: 1**	
			Core C: 3	**Core A: 1** **Core B: 1**
				Core C: 1

No pipline:
total cycle counts = 6+4+1+3+1+1+2+4+1+3+1+1+1 = 29

With pipeline:
total cycle counts = 6+4+2+4+1 = 17

17 > 29/2 -> It is impossible to achieve a speedup of 2.



```
00000000000104ac:   j 0x104b8 <fast_power_iter+16>
12                      x = x * x;
00000000000104b0:   mulw a5,a5,a5
13                      pow >>= 1;
00000000000104b4:   sraiw a1,a1,0x1
6                   while(pow)
00000000000104b8:   c.beqz a1,0x104c4 <fast_power_iter+
8                       if(pow & 1)
00000000000104ba:   bbc a1,0,0x104b0 <fast_power_iter+8:
10                          res = res * x;
00000000000104be:   mulw a0,a5,a0
00000000000104c2:   c.j 0x104b0 <fast_power
```

Name	Calls	Self InsC	Self CycC	Total InsC	Total CycC	Time Percent...
memset	1	26	38	35	51	39.18%
_start	1	23	32	69	97	32.99%
memset +	1	9	13	9	13	13.40%
main	1	6	8	11	14	8.25%
fast_powe	1	5	6	5	6	6.19%

Core C: 6 cycle counts

```
12                      x = x * x;
00000000000104b0:   mulw a5,a5,a5
13                      pow >>= 1;
00000000000104b4:   sraiw a1,a1,0x1
6                   while(pow)
00000000000104b8:   c.beqz a1,0x104c4 <fast_power_iter+
8                       if(pow & 1)
00000000000104ba:   bbc a1,0,0x104b0 <fast_power_iter+8:
10                          res = res * x;
00000000000104be:   mulw a0,a5,a0
00000000000104c2:   c.j 0x104b0 <fast_power_iter+8>
```

Name	Calls	Self InsC	Self CycC	Total InsC	Total CycC	Time Percent...
memset	1	26	38	35	51	36.89%
_start	1	23	32	73	103	31.07%
memset +	1	9	13	9	13	12.62%
fast_powe	1	9	12	9	12	11.65%
main	1	6	8	15	20	7.77%

```
12                      x = x * x;
00000000000104b0:   mulw a5,a5,a5
13                      pow >>= 1;
00000000000104b4:   sraiw a1,a1,0x1
6                   while(pow)
00000000000104b8:   c.beqz a1,0x104c4 <fast_power_iter+
8                       if(pow & 1)
00000000000104ba:   bbc a1,0,0x104b0 <fast_power_iter+8:
10                          res = res * x;
00000000000104be:   mulw a0,a5,a0
00000000000104c2:   c.j 0x104b0 <fast_power_iter+8>
```

Total Self Cycle Count: 104

Name	Calls	Self InsC	Self CycC	Total InsC	Total CycC	Time Percent...
memset	1	26	38	35	51	36.54%
_start	1	23	32	74	104	30.77%
memset +	1	9	13	9	13	12.50%
fast_powe	1	10	13	10	13	12.50%
main	1	6	8	16	21	7.69%

Core A: 4 cycle counts

```
12                      x = x * x;
00000000000104b0:   mulw a5,a5,a5
13                      pow >>= 1;
00000000000104b4:   sraiw a1,a1,0x1
6                   while(pow)
00000000000104b8:   c.beqz a1,0x104c4 <fast_power_iter+
8                       if(pow & 1)
00000000000104ba:   bbc a1,0,0x104b0 <fast_power_iter+8:
10                          res = res * x;
00000000000104be:   mulw a0,a5,a0
00000000000104c2:   c.j 0x104b0 <fast_power_iter+8>
```

Name	Calls	Self InsC	Self CycC	Total InsC	Total CycC	Time Percent...
memset	1	26	38	35	51	35.19%
_start	1	23	32	75	108	29.63%
fast_powe	1	11	17	11	17	15.74%
memset +	1	9	13	9	13	12.04%
main	1	6	8	17	25	7.41%

Core B: 1 cycle counts

```
12                      x = x * x;
00000000000104b0:   mulw a5,a5,a5
13                      pow >>= 1;
00000000000104b4:   sraiw a1,a1,0x1
6                   while(pow)
00000000000104b8:   c.beqz a1,0x104c4 <fast_power_iter+
8                       if(pow & 1)
00000000000104ba:   bbc a1,0,0x104b0 <fast_power_iter+8:
10                          res = res * x;
00000000000104be:   mulw a0,a5,a0
00000000000104c2:   c.j 0x104b0 <fast_power
```

Name	Calls	Self InsC	Self CycC	Total InsC	Total CycC	Time Percent...
memset	1	26	38	35	51	34.86%
_start	1	23	32	76	109	29.36%
fast_powe	1	12	18	12	18	16.51%
memset +	1	9	13	9	13	11.93%
main	1	6	8	18	26	7.34%

Core C: 3 cycle counts

```
12                      x = x * x;
00000000000104b0:   mulw a5,a5,a5
13                      pow >>= 1;
00000000000104b4:   sraiw a1,a1,0x1
6                   while(pow)
00000000000104b8:   c.beqz a1,0x104c4 <fast_power_iter+
8                       if(pow & 1)
00000000000104ba:   bbc a1,0,0x104b0 <fast_power_iter+8:
10                          res = res * x;
00000000000104be:   mulw a0,a5,a0
00000000000104c2:   c.j 0x104b0 <fast_power_iter+8>
```

Total Self Cycle Count: 112

Name	Calls	Self InsC	Self CycC	Total InsC	Total CycC	Time Percent...
memset	1	26	38	35	51	33.93%
_start	1	23	32	80	112	28.57%
fast_powe	1	16	21	16	21	18.75%
memset +	1	9	13	9	13	11.61%
main	1	6	8	22	29	7.14%

```
12                      x = x * x;
00000000000104b0:   mulw a5,a5,a5
13                      pow >>= 1;
00000000000104b4:   sraiw a1,a1,0x1
6                   while(pow)
00000000000104b8:   c.beqz a1,0x104c4 <fast_power_iter+
8                       if(pow & 1)
00000000000104ba:   bbc a1,0,0x104b0 <fast_power_iter+8:
10                          res = res * x;
00000000000104be:   mulw a0,a5,a0
00000000000104c2:   c.j 0x104b0 <fast_power_
```

Total Self Cycle Count: 113

Name	Calls	Self InsC	Self CycC	Total InsC	Total CycC	Time Percent...
memset	1	26	38	35	51	33.63%
_start	1	23	32	81	113	28.32%
fast_powe	1	17	22	17	22	19.47%
memset +	1	9	13	9	13	11.50%
main	1	6	8	23	30	7.08%

Core A: 1 cycle counts

```
12                      x = x * x;
00000000000104b0:   mulw a5,a5,a5
13                      pow >>= 1;
00000000000104b4:   sraiw a1,a1,0x1
6                   while(pow)
00000000000104b8:   c.beqz a1,0x104c4 <fast_power_iter+
8                       if(pow & 1)
00000000000104ba:   bbc a1,0,0x104b0 <fast_power_iter+8:
10                          res = res * x;
00000000000104be:   mulw a0,a5,a0
00000000000104c2:   c.j 0x104b0 <fast_power_iter+8>
```

Total Self Cycle Count: 114

Name	Calls	Self InsC	Self CycC	Total InsC	Total CycC	Time Percent...
memset	1	26	38	35	51	33.33%
_start	1	23	32	82	114	28.07%
fast_powe	1	18	23	18	23	20.18%
memset +	1	9	13	9	13	11.40%
main	1	6	8	24	31	7.02%

```
12                 x = x * x;
00000000000104b0:  mulw a5,a5,a5
13                 pow >>= 1;
00000000000104b4:  sraiw a1,a1,0x1
6               while(pow)
00000000000104b8:  c.beqz a1,0x104c4 <fast_power_iter+:
8                  if(pow & 1)
00000000000104ba:  bbc a1,0,0x104b0 <fast_power_iter+8:
10                   res = res * x;
00000000000104be:  mulw a0,a5,a0
00000000000104c2:  c.j 0x104b0 <fast_power_
```

Core B: 1 cycle counts

Name	Calls	Self InsC	Self CycC	Total InsC	Total CycC	Time Percent...
memset	1	26	38	35	51	33.04%
_start	1	23	32	83	115	27.83%
fast_powe	1	19	24	19	24	20.87%
memset +	1	9	13	9	13	11.30%
main	1	6	8	25	32	6.96%

```
12                 x = x * x;
00000000000104b0:  mulw a5,a5,a5
13                 pow >>= 1;
00000000000104b4:  sraiw a1,a1,0x1
6               while(pow)
00000000000104b8:  c.beqz a1,0x104c4 <fast_power_iter+:
8                  if(pow & 1)
00000000000104ba:  bbc a1,0,0x104b0 <fast_power_iter+8:
10                   res = res * x;
00000000000104be:  mulw a0,a5,a0
00000000000104c2:  c.j 0x104b0 <fast_power_
```

Core C: 2 cycle counts

Total Self Cycle Count: 117

Name	Calls	Self InsC	Self CycC	Total InsC	Total CycC	Time Percent...
memset	1	26	38	35	51	32.48%
_start	1	23	32	86	117	27.35%
fast_powe	1	22	26	22	26	22.22%
memset +	1	9	13	9	13	11.11%
main	1	6	8	28	34	6.84%

```
12                 x = x * x;
00000000000104b0:  mulw a5,a5,a5
13                 pow >>= 1;
00000000000104b4:  sraiw a1,a1,0x1
6               while(pow)
00000000000104b8:  c.beqz a1,0x104c4 <fast_power_iter+:
8                  if(pow & 1)
00000000000104ba:  bbc a1,0,0x104b0 <fast_power_iter+8:
10                   res = res * x;
00000000000104be:  mulw a0,a5,a0
00000000000104c2:  c.j 0x104b0 <fast_power_
```

Core A: 4 cycle counts

Total Self Cycle Count: 121

Name	Calls	Self InsC	Self CycC	Total InsC	Total CycC	Time Percent...
memset	1	26	38	35	51	31.40%
_start	1	23	32	87	121	26.45%
fast_powe	1	23	30	23	30	24.79%
memset +	1	9	13	9	13	10.74%
main	1	6	8	29	38	6.61%

```
12                 x = x * x;
00000000000104b0:  mulw a5,a5,a5
13                 pow >>= 1;
00000000000104b4:  sraiw a1,a1,0x1
6               while(pow)
00000000000104b8:  c.beqz a1,0x104c4 <fast_power_iter+:
8                  if(pow & 1)
00000000000104ba:  bbc a1,0,0x104b0 <fast_power_iter+8:
10                   res = res * x;
00000000000104be:  mulw a0,a5,a0
00000000000104c2:  c.j 0x104b0 <fast_power_
```

Core B: 1 cycle counts

Name	Calls	Self InsC	Self CycC	Total InsC	Total CycC	Time Percent...
memset	1	26	38	35	51	31.15%
_start	1	23	32	88	122	26.23%
fast_powe	1	24	31	24	31	25.41%
memset +	1	9	13	9	13	10.66%
main	1	6	8	30	39	6.56%

```
12                 x = x * x;
00000000000104b0:  mulw a5,a5,a5
13                 pow >>= 1;
00000000000104b4:  sraiw a1,a1,0x1
6               while(pow)
00000000000104b8:  c.beqz a1,0x104c4 <fast_power_iter+:
8                  if(pow & 1)
00000000000104ba:  bbc a1,0,0x104b0 <fast_power_iter+8:
10                   res = res * x;
00000000000104be:  mulw a0,a5,a0
00000000000104c2:  c.j 0x104b0 <fast_power_iter+8>
```

Core C: 3 cycle counts

Total Self Cycle Count: 125

Name	Calls	Self InsC	Self CycC	Total InsC	Total CycC	Time Percent...
memset	1	26	38	35	51	30.40%
fast_powe	1	28	34	28	34	27.20%
_start	1	23	32	92	125	25.60%
memset +	1	9	13	9	13	10.40%
main	1	6	8	34	42	6.40%

```
12                 x = x * x;
00000000000104b0:  mulw a5,a5,a5
13                 pow >>= 1;
00000000000104b4:  sraiw a1,a1,0x1
6               while(pow)
00000000000104b8:  c.beqz a1,0x104c4 <fast_power_iter+:
8                  if(pow & 1)
00000000000104ba:  bbc a1,0,0x104b0 <fast_power_iter+8:
10                   res = res * x;
00000000000104be:  mulw a0,a5,a0
00000000000104c2:  c.j 0x104b0 <fast_power_
```

Total Self Cycle Count: 126

Name	Calls	Self InsC	Self CycC	Total InsC	Total CycC	Time Percent...
memset	1	26	38	35	51	30.16%
fast_powe	1	29	35	29	35	27.78%
_start	1	23	32	93	126	25.40%
memset +	1	9	13	9	13	10.32%
main	1	6	8	35	43	6.35%

```
12                 x = x * x;
00000000000104b0:  mulw a5,a5,a5
13                 pow >>= 1;
00000000000104b4:  sraiw a1,a1,0x1
6               while(pow)
00000000000104b8:  c.beqz a1,0x104c4 <fast_power_iter+:
8                  if(pow & 1)
00000000000104ba:  bbc a1,0,0x104b0 <fast_power_iter+8:
10                   res = res * x;
00000000000104be:  mulw a0,a5,a0
00000000000104c2:  c.j 0x104b0 <fast_power_iter+8>
```

Core A: 1 cycle counts

Name	Calls	Self InsC	Self CycC	Total InsC	Total CycC	Time Percent...
memset	1	26	38	35	51	29.92%
fast_powe	1	30	36	30	36	28.35%
_start	1	23	32	94	127	25.20%
memset +	1	9	13	9	13	10.24%
main	1	6	8	36	44	6.30%

```
12                 x = x * x;
00000000000104b0:  mulw a5,a5,a5
13                 pow >>= 1;
00000000000104b4:  sraiw a1,a1,0x1
6               while(pow)
00000000000104b8:  c.beqz a1,0x104c4 <fast_power_iter+:
8                  if(pow & 1)
00000000000104ba:  bbc a1,0,0x104b0 <fast_power_iter+8:
10                   res = res * x;
00000000000104be:  mulw a0,a5,a0
00000000000104c2:  c.j 0x104b0 <fast_power_iter+8>
```

Core B: 1 cycle counts

Name	Calls	Self InsC	Self CycC	Total InsC	Total CycC	Time Percent...
memset	1	26	38	35	51	29.69%
fast_powe	1	31	37	31	37	28.91%
_start	1	23	32	95	128	25.00%
memset +	1	9	13	9	13	10.16%
main	1	6	8	37	45	6.25%

Core C: 1 cycle counts

```
00000000000104b0:   mulw a5,a5,a5
13                          pow >>= 1;
00000000000104b4:   sraiw a1,a1,0x1
 6                    while(pow)
00000000000104b8:   c.beqz a1,0x104c4 <fast_power_iter+2
 8                          if(pow & 1)
00000000000104ba:   bbc a1,0,0x104b0 <fast_power_iter+8>
10                              res = res * x;
00000000000104be:   mulw a0,a5,a0
00000000000104c2:   c.j 0x104b0 <fast_power_iter+8>
16                  }
00000000000104c4:   ret
```

Total Self Cycle Count: 129

Name	Calls	Self InsC	Self CycC	Total InsC	Total CycC	Time Percent...
memset	1	26	38	35	51	29.46%
fast_powe	1	32	38	32	38	29.46%
_start	1	23	32	96	129	24.81%
memset +	1	9	13	9	13	10.08%
main	1	6	8	38	46	6.20%

2. (30 points) ***RISC-V Assembly Code***

Consider a little-endian 64-bit RISC-V sequential processor with the following contents in the register set, data memory, and code memory. Assume that the current **PC** has the value **0x0000 0000 0001 00B0**.

Reg.	Initial value	Reg.	Initial value	Memory address	Initial value
x0	0x0000 0000 0000 0000	x16	0x0000 0000 0000 0004	0x0000 003E FF20 13C0	0x0000 0055
x1	0x0000 0000 0001 00B0	x17	0x0000 0000 0000 0020	0x0000 003E FF20 13C4	0x0000 0000
x2	0x0000 003E FF20 13E0	x18	0x0000 0000 0000 0003	0x0000 003E FF20 13C8	0x0A0C 0345
x3	0x0000 0000 0001 0000	x19	0x0000 0000 0000 0040	0x0000 003E FF20 13CC	0x0450 0000
x4	0x0000 003E FF20 13C0	x20	0x0000 0000 0000 0000	0x0000 003E FF20 13D0	0x000D 0000
x5	0x0000 0000 0000 0008	x21	0x0000 0000 0000 0000	0x0000 003E FF20 13D4	0x0A00 0010
x6	0x0000 0000 0000 0004	x22	0x1111 FFFF 0000 5555	0x0000 003E FF20 13D8	0x0020 0000
x7	0x0000 0000 0000 003A	x23	0x0000 0000 0000 0000	0x0000 003E FF20 13DC	0x4000 0000
x8	0x0000 003E FF20 1400	x24	0x0000 0000 0000 0000	0x0000 003E FF20 13E0	0x8000 A000
x9	0x0000 0000 0000 0007	x25	0x0000 0000 0000 0034	0x0000 003E FF20 13E4	0xA800 3F10
x10	0x0000 0000 0000 0050	x26	0x0000 003E FF20 13F0	0x0000 003E FF20 13E8	0x0091 0000
x11	0x0000 003E FF20 1530	x27	0x0000 003E FF20 13FC	0x0000 003E FF20 13EC	0x0000 0000
x12	0x0000 0000 0000 1AF3	x28	0x0000 FFFF 0000 FFFF	0x0000 003E FF20 13F0	0x00C1 A000
x13	0x0000 0000 0000 0000	x29	0x0000 0000 0000 000A	0x0000 003E FF20 13F4	0x0130 00F0
x14	0x0000 F94E 17CC B154	x30	0x0000 0000 00F0 0000	0x0000 003E FF20 13F8	0x0041 0000
x15	0xCCCC 0000 0000 0000	x31	0x0000 00F0 0000 0000	0x0000 003E FF20 13FC	0x0A0B 0130

Note: for convenience, there is a unique instruction ID for each instruction in the code memory.

Instruction ID	Label	Code address	Instruction
1		0x0000 0000 0001 00B0	`sub x7, x5, x6`
2		0x0000 0000 0001 00B4	`sd x2, 16(x2)`
3		0x0000 0000 0001 00B8	`jal x1, BEGIN`
4		0x0000 0000 0001 00BC	`lw x6, 4(x2)`
5		0x0000 0000 0001 00C0	`0x0040 8067`
6	**BEGIN:**	0x0000 0000 0001 00C4	`0xFF84 3283`
7		0x0000 0000 0001 00C8	`and x5, x30, x5`
8		0x0000 0000 0001 00CC	`0x4142 D293`
9		0x0000 0000 0001 00D0	`add x0, x5, x7`
10		0x0000 0000 0001 00D4	`add x28, x0, x2`
11		0x0000 0000 0001 00D8	`lb x7, 10(x28)`
12		0x0000 0000 0001 00DC	`bge x7, x29, END`
13		0x0000 0000 0001 00E0	`jalr x1, 0(x1)`
14	**END:**	0x0000 0000 0001 00E4	`addi x7, x7, -16`
15		0x0000 0000 0001 00E8	`xor x7, x6, x7`
16		0x0000 0000 0001 00EC	`srai x7, x7, 16`
17		0x0000 0000 0001 00F0	`addi x31, x6, 1000`
18		0x0000 0000 0001 00F4	`srai x31, x31, 16`
19		0x0000 0000 0001 00F8	`sb x5, -8(x8)`
20		0x0000 0000 0001 00FC	`sd x31, -24(x8)`

(a) (5 points) Decode instructions with instruction IDs 5, 6, and 8. Then, briefly explain their functionalities.

Instruction ID	Hexadecimal Encoded instruction	Decoded instruction and brief explanation
5	`0x0040 8067`	`jalr x0,4(x1)` 跳回 x1+4 那個 address 存的指令 然後將下一行的 address 放進 x0 (但 x0 永遠為 0)
6	`0xFF84 3283`	`ld x5,-8(x8)` 將 x8-8 所存的 double word load 進 x5
8	`0x4142 D293`	`srai x5,x5,20` 將 x5 做算術右移 20 位

(b) (15 points) Trace the execution flow of the assembly code. Extend the table below. For each executed instruction, record its ID, any updated registers and/or memory cells, and their new values (if any) in hexadecimal representation. Annotate updated memory values per 32-bit word. The first three instructions have been completed for you.

Instruction ID	Updated register	Updated memory
1	`x7 <- x5-x6 = 0x0000 0000 0000 0004`	
2		`MEM[0x0000 003E FF20 13F0] <- 0xFF20 13E0` `MEM[0x0000 003E FF20 13F4] <- 0x0000 003E`
3	`x1 <- PC+4 = 0x0000 0000 0001 00BC`	
6	`x5 <- -8(x8) = 0x0A0B 0130 0041 0000`	
7	`x5 <- x5&x30 = 0x0000 0000 0040 0000`	
8	`x5 <- x5>>20 = 0x0000 0000 0000 0004`	
9		
10	`x28 <- x0+x2 = 0x0000 003E FF20 13E0`	
11	`x7 <- 10(x28) 0xFFFF FFFF FFFF FF91`	
12		
13	`x1 <- PC+4 = 0x0000 0000 0001 00E4`	
4	`x6 <- 4(x2) = 0xFFFF FFFF A800 3F10`	
5		
[redacted]	[redacted]	
15	`x7 <- x6^x7 = 0x0000 0000 57FF C081`	
16	`x7 <- x7>>16 = 0x0000 0000 0000 57FF`	
17	`x31 <- x6+1000 = 0xFFFF FFFF A800 42F8`	
18	`x31 <- x31>>16 = 0xFFFF FFFF FFFF A800`	
19		`MEM[0x0000 003E FF20 13F8] <- 0x0041 0004`
20		`MEM[0x0000 003E FF20 13E8] <- 0xFFFF A800` `MEM[0x0000 003E FF20 13EC] <- 0xFFFF FFFF`

(c) (5 points) Once you have completed the execution flow table, count the total number of memory accesses (excluding register accesses) performed throughout the code.

共 6 次

(d) (5 points) Suppose you want to insert an instruction after instruction ID 20. This instruction should use a **blt** to jump to the label **BEGIN** if the value in register **x31** is less than **x7**. Complete the table below with the instruction. Show how you convert the Assembly instruction into its hexadecimal representation. Moreover, will the branch be taken?

Code address	Assembly instruction	Hexadecimal encoded instruction	Taken?
0x0000 0000 0001 0100	**blt x31,x7,BEGIN**	**0xFC7FC2E3**	**Yes**

opcode: 1100011

imm[4:1,11]: 0010 1

func3: 100

rs1: 11111

rs2: 00111

imm[12,10:5]: 1 111110

(imm: 1 1111 1100 0100)

instruction: 1 111110 00111 11111 100 0010 1 1100011

3. (10 points) ***RISC-V Assembly to C***

Translate the following RISC-V Assembly code to the equivalent C code. Indicate the corresponding C code for each line of Assembly. Assume that variables, **`m`**, **`i`**, **`j`**, and **`total`** are stored in registers **`x3`**, **`x10`**, **`x11`**, and **`x12`**, respectively. **`MemArray`** is an array (consisting of 4-byte integers as its elements) with its base address stored in register **`x13`**.

```
      addi x10, x0, 0
      addi x28, x13, 0
LOOPI:
      bge x10, x3, ENDI
      addi x11, x0, 0
      addi x12, x0, 0
      lw x29, 0(x28)
      addi x30, x0, 32
LOOPJ:
      bge x11, x30, ENDJ
      srl x31, x29, x11
      andi x31, x31, 1
      add x12, x12, x31
      addi x11, x11, 1
      jal x0, LOOPJ
ENDJ:
      sw x12, 0(x28)
      addi x10, x10, 1
      addi x28, x28, 4
      jal x0, LOOPI
ENDI:
```

1	`addi x10, x0, 0`	1	`i = 0`
2	`addi x28, x13, 0`	2	`// set x28 point to MemArray[0]`
	`LOOPI:`		
3	`bge x10, x3, ENDI`	3	`while (i < m) {`
4	`addi x11, x0, 0`	4	`j = 0;`
5	`addi x12, x0, 0`	5	`total = 0;`
6	`lw x29, 0(x28)`	6	`// load MemArray[i] to x29`
7	`addi x30, x0, 32`	7	`// set x30 = 32`
	`LOOPJ:`		
8	`bge x11, x30, ENDJ`	8	`while (j < 32) {`
9	`srl x31, x29, x11`	9	`unsigned int tmp = MemArray[i];`
			`// 因為是 srl 所以要先存成 unsigned, tmp >> j`
10	`andi x31, x31, 1`	10	`// (tmp >> j) & 1`
11	`add x12, x12, x31`	11	`total += (tmp >> j) & 1;`
12	`addi x11, x11, 1`	12	`j++;`
13	`jal x0, LOOPJ`	13	`}`
	`ENDJ:`		
14	`sw x12, 0(x28)`	14	`MemArray[i] = total;`
15	`addi x10, x10, 1`	15	`i++;`
16	`addi x28, x28, 4`	16	`// 同為上一行的 i++`
17	`jal x0, LOOPI`	17	`}`
	`ENDI:`		

4. (10 points) ***C to RISC-V Assembly***

For the following C statement, write the corresponding RISC-V Assembly code. Assume that the base addresses of arrays **`A`** and **`B`** are in registers **`x5`** and **`x6`**, respectively, and the variables **`i`** and **`j`** are assigned to registers **`x7`** and **`x11`**, respectively.

```
j = B[A[i*4 + 1]] + B[i]
```

(a) (5 points) Assume that the elements of the arrays **`A`** and **`B`** are 4-byte words.

```
slli x28,x7,2      # 將 x28 設成 i*4
add  x29,x28,x6    # 將 x29 指到 B[i]
lw   x31,0(x29)    # 將 B[i] load 進 x31
addi x28,x28,1     # 將 x28 設成 (i*4) + 1
slli x28,x28,2     # 因為是 4-byte word，所以將 x28 * 4 算出 offset
add  x28,x28,x5    # 將 x28 指到 A[i*4 + 1]
lw   x29,0(x28)    # 將 A[i*4 + 1] load 進 x29
slli x29,x29,2     # 因為是 4-byte word，所以將 x29 * 4 算出 offset
add  x29,x29,x6    # 將 x29 指到 B[A[i*4 + 1]]
lw   x30,0(x29)    # 將 B[A[i*4 + 1]] load 進 x30
add  x11,x30,x31   # 將 x11 設成 B[A[i*4 + 1]] + B[i]
```

(b) (5 points) Assume that the elements of the arrays **`A`** and **`B`** are 8-byte words.

```
slli x28,x7,2      # 將 x28 設成 i*4
addi x28,x28,1     # 將 x28 設成 i*4 + 1
slli x28,x28,3     # 因為是 8-byte word，所以將 x28 * 8 算出 offset
add  x28,x28,x5    # 將 x28 指到 A[i*4 + 1]
ld   x29,0(x28)    # 將 A[i*4 + 1] load 進 x29
slli x29,x29,3     # 因為是 8-byte word，所以將 x29 * 8 算出 offset
slli x28,x7,3      # 因為是 8-byte word，所以將 x7 * 8 算出 offset
add  x29,x29,x6    # 將 x29 指到 B[A[i*4 + 1]]
add  x28,x28,x6    # 將 x28 指到 B[i]
ld   x30,0(x29)    # 將 B[A[i*4 + 1]] load 進 x30
ld   x31,0(x28)    # 將 B[i] load 進 x31
add  x11,x30,x31   # 將 x11 設成 B[A[i*4 + 1]] + B[i]
```

5. (15 points) ***RISC-V Calling Convention***

Implement the following C code in RISC-V Assembly. Note the RISC-V Spec: “In the standard RISC-V calling convention, the stack grows downward and the stack pointer is always kept 16-byte aligned.” Moreover, write down comments to describe the Assembly code clearly.

```
long long int Func(int n)
     {if (n == 0) {
          return 0;
     }

     if ((n & 1) != 0) {
          return n + Func(n >> 1);
     } else {
          return Func(n >> 1);
     }
}
```

```
Func:
    addi x2,x2,-16      # 將 x2 往下移 16
    sd   x1,8(x2)       # 把 x1 的值存進 8(x2)
    sd   x10,0(x2)      # 把 n 的值存進 0(x2)
    bne  x10,x0,Odd     # 若 n != 0 則跳至 Odd
    addi x10,x0,0       # 將 x10 設成 0
    addi x2,x2,16       # 將 x2 往上移 16
    jalr x0,0(x1)       # return 回去上一層 function
Odd:
    andi x28,x10,1      # x28 <- n & 1
    beq  x28,x0,Even    # 若 (n & 1) == 0 則跳到 Even
    srai x10,x10,1      # x10 <- n >> 1
    jal  x1,Func        # 呼叫 Func(n >> 1)
    ld   x29,0(x2)      # 將 0(x2) 的值 load 到 x29 (原本這層 n 的值)
    ld   x1,8(x2)       # 將 8(x2) 的值 load 回 x1 (以 return 回正確的地方)
    add  x10,x10,x29    # 將 return value 設成 n + Func(n >> 1)
    addi x2,x2,16       # 將 x2 往上移 16
    jalr X0,0(x1)       # return 回去上一層 function

Even:
    srai x10,x10,1      # x10 <- n >> 1
    jal  x1,Func        # 呼叫 Func(n >> 1)
    ld   x1,8(x2)       # 將 8(x2) 的值 load 回 x1 (以 return 回正確的地方)
    addi x2,x2,16       # 將 x2 往上移 16
    jalr x0,0(x1)       # return 回去上一層 function
```