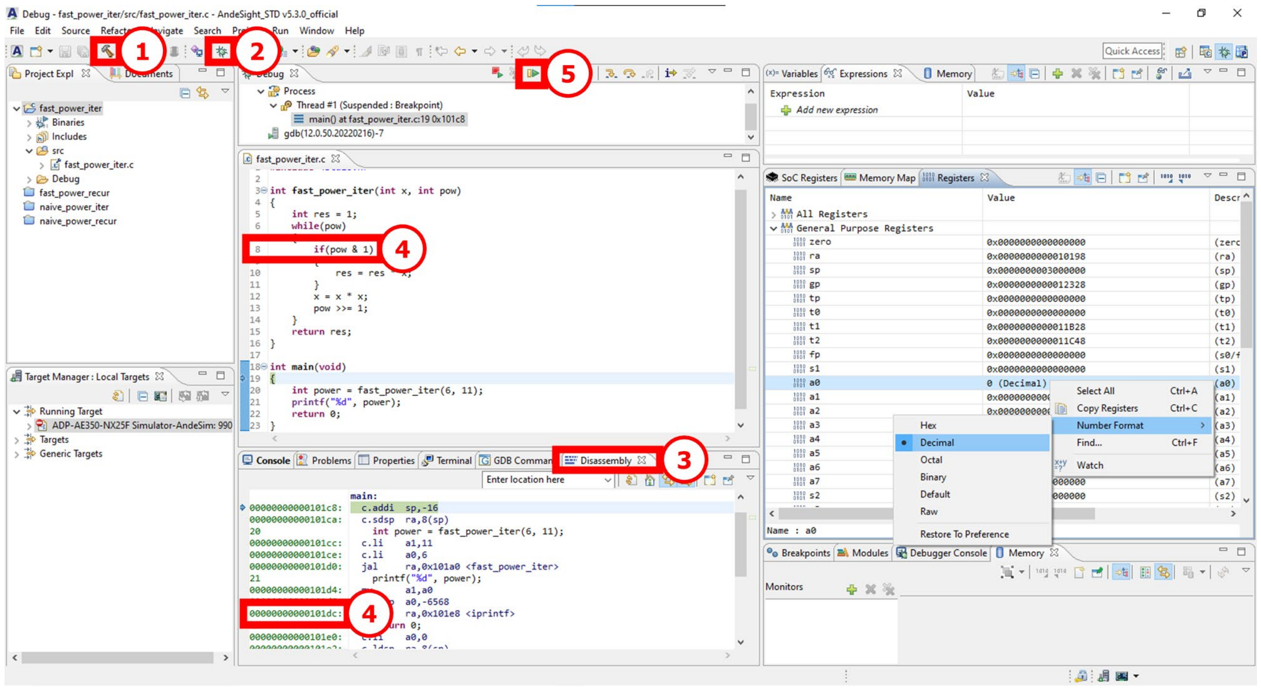
Department of Computer Science
National Tsing Hua University
EECS403000 Computer Architecture
Spring 2024, Homework 2
Due date: **April 11, 2024 23:59 pm**

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.



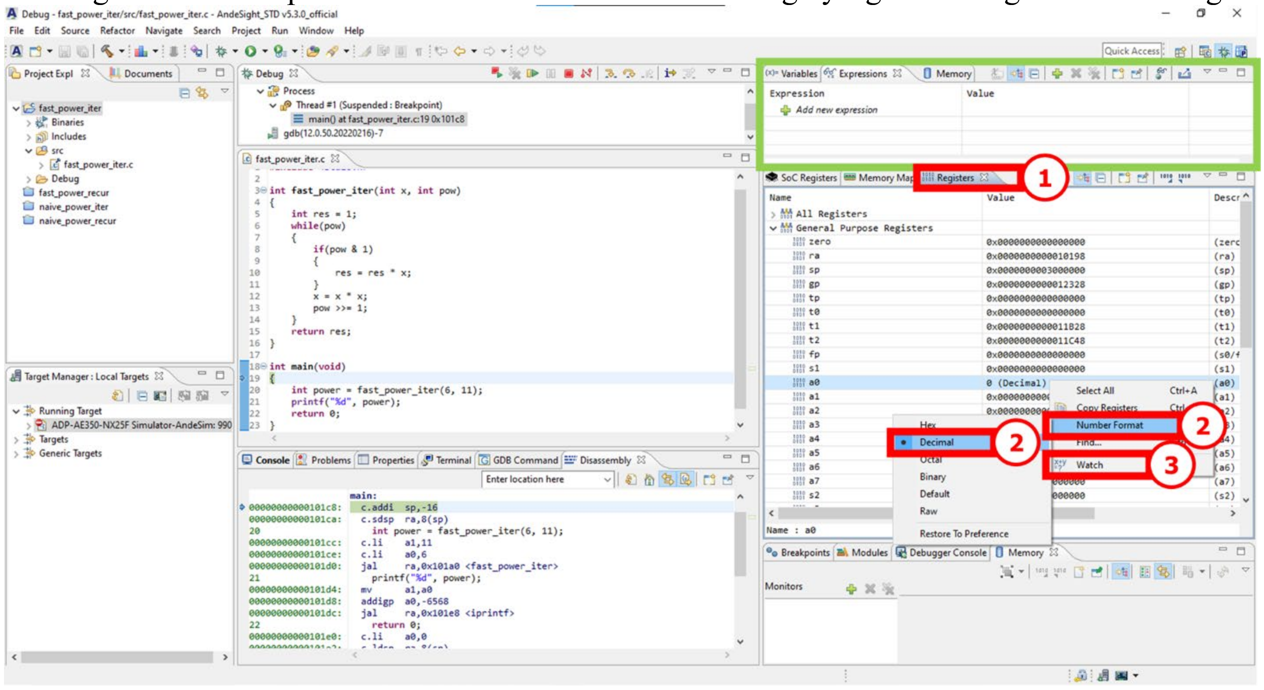
```c
int fast_power_iter(int x, int pow)
{
    int res = 1;
    while(pow)
    {
        if(pow & 1)
        {
            res = res * x;
        }
        x = x * x;
        pow >>= 1;
    }
    return res;
}

int main(void)
{
    int power = fast_power_iter(6, 11);
    printf("%d", power);
    return 0;
}
```

```
                    main:
00000000000101c8:    c.addi  sp,-16
00000000000101ca:    c.sdsp  ra,8(sp)
20                     int power = fast_power_iter(6, 11);
00000000000101cc:    c.li    a1,11
00000000000101ce:    c.li    a0,6
00000000000101d0:    jal     ra,0x101a0 <fast_power_iter>
21                     printf("%d", power);
00000000000101d4:            a1,a0
                             a0,-6568
00000000000101dc:            ra,0x101e8 <iprintf>
                          urn 0;
00000000000101e0:            a0,0
```

Name	Value	Descr
All Registers		
General Purpose Registers		
zero	0x0000000000000000	(zerc
ra	0x0000000000010198	(ra)
sp	0x0000000003000000	(sp)
gp	0x0000000000012328	(gp)
tp	0x0000000000000000	(tp)
t0	0x0000000000000000	(t0)
t1	0x0000000000011B28	(t1)
t2	0x0000000000011C48	(t2)
fp	0x0000000000000000	(s0/f
s1	0x0000000000000000	(s1)
a0	0 (Decimal)	(a0)
a1	0x000000000	(a1)
a2	0x000000000	(a2)
a3		(a3)
a4		(a4)
a5		(a5)
a6		(a6)
a7	000000	(a7)
s2	000000	(s2)

Name : a0

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.



```
                    main:
00000000000101c8:    c.addi  sp,-16
00000000000101ca:    c.sdsp  ra,8(sp)
20                     int power = fast_power_iter(6, 11);
00000000000101cc:    c.li    a1,11
00000000000101ce:    c.li    a0,6
00000000000101d0:    jal     ra,0x101a0 <fast_power_iter>
21                     printf("%d", power);
00000000000101d4:    mv      a1,a0
00000000000101d8:    addigp  a0,-6568
00000000000101dc:    jal     ra,0x101e8 <iprintf>
22                     return 0;
00000000000101e0:    c.li    a0,0
```

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`			
`fast_power_iter`			

(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

(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`						
`fast_power_recur`						

(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.

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`	
6	`0xFF84 3283`	
8	`0x4142 D293`	

(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`	

(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.

(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?

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:
```

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

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);
      }
}
```