Department of Computer Science, National Tsing Hua University

CS4100 Computer Architecture

Spring 2025, Homework 2

Due Date: 31st March 2025 23:59

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

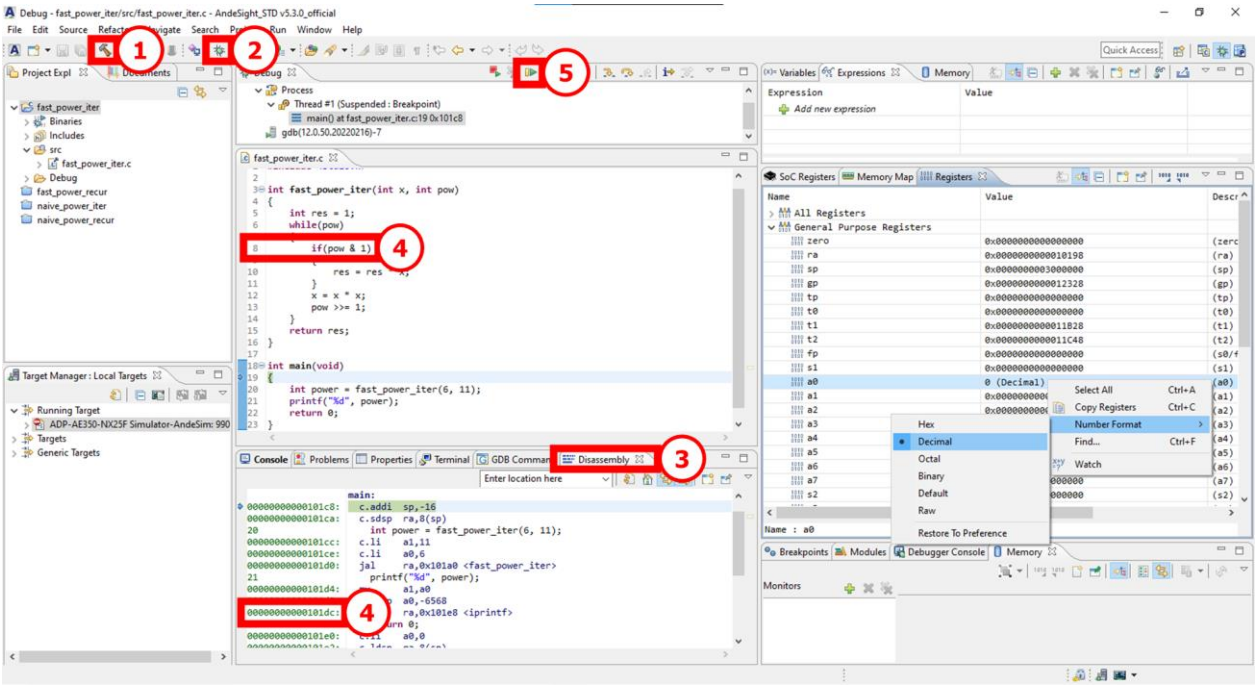
(1) Build the program with the Debug configuration.

(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.

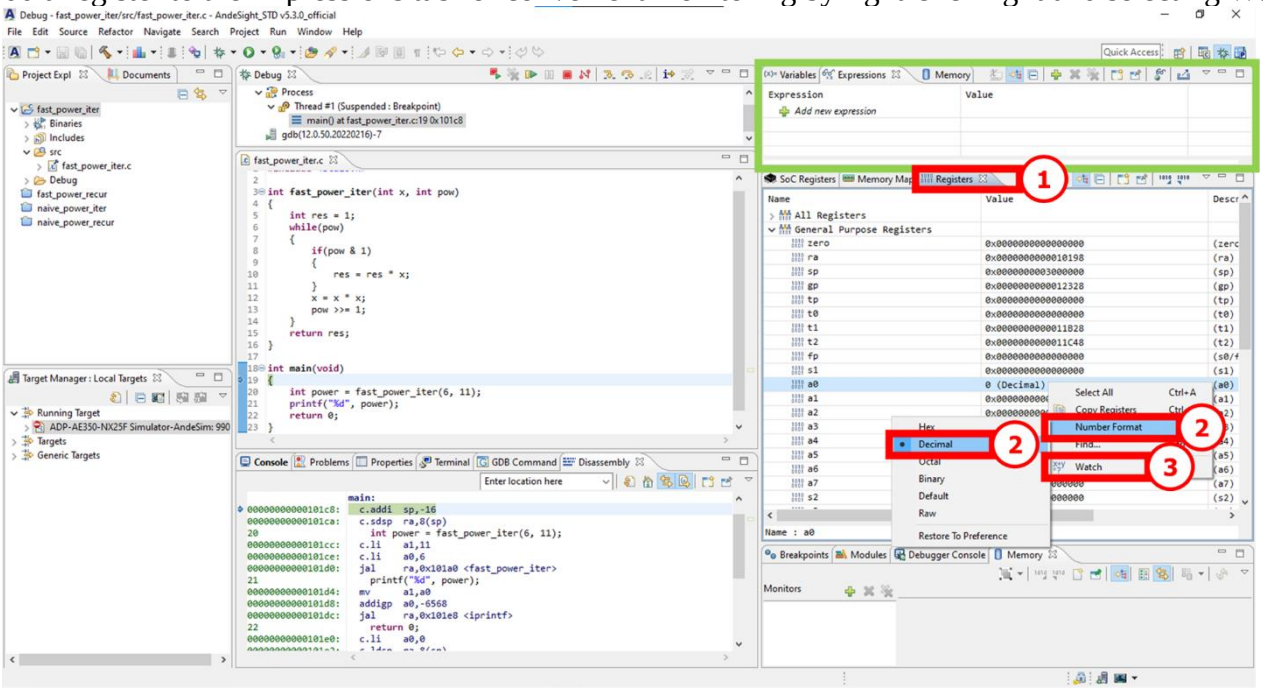


To observe register value changes, Start Debugging as an Application Program and follow the steps:

(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. (40%) ***AndeSight***

This question explores the Fibonacci Algorithm implemented in iterative and recursive manners with AndeSight™ with a setup similar to Homework 1. We will analyze the source code for **`iter_fib.c`** and **`recur_fib.c`**. The default optimization is **`-Og`** by default, unless stated otherwise.

(a) (6%) ***RISC-V Calling Convention***

Compilers typically translate functions into subroutine and perform function calls using a jump instruction following a call convention. While function calls can be inefficient, they are essential in programming. Complete Table 1 by locating the starting and ending memory addresses of the code memory allocated to the **`iter_fibonacci`** and **`recur_fibonacci`** subroutines by examining the Assembly code, and identifying how these subroutines are called within **`main()`** and write the instructions down in the Reference field in Table 1.

Table 1. Addresses and references of **`iter_fibonacci`** and **`recur_fibonacci`** subroutines.

Subroutine Name	Starting Memory Address	Ending Memory Address	Reference
`iter_fibonacci`			
`recur_fibonacci`			

(b) (6%) ***RISC-V Calling Convention***

To make common cases fast, compilers allocate application program variable to processor registers, which are much faster than memory. Compile both **`iter_fib.c`** and **`recur_fib.c`** programs with **`-Og`** optimization flag. Complete Table 2 by answering the following questions:

(i) When calling **`iter_fibonacci`** and **`recur_fibonacci`** from **`main()`**, what registers hold the argument **`num`** and what values do they contain?

(ii) After **`iter_fibonacci`** and **`recur_fibonacci`** return to **`main()`**, what register stores the return value, and what is its value?

Table 2. Registers used for arguments and return values.

Function Name	Argument **`num`**		Return Value	
	Register Name	Value	Register Name	Value
`iter_fibonacci`				
`recur_fibonacci`				

(c) (8%) ***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 **`recur_fibonacci`** function.

Find and record the instructions and the corresponding memory addresses that save these registers in Table 3. Furthermore, Record the register names and their corresponding stack offsets. You may need to extend the number of rows in Table 3.

Note: Order the table by decreasing memory addresses in which these registers are being saved.

Table 3. Stack layout in **`recur_fibonacci`**.

Memory Address	Instruction	Saved Registers	Stack Offset

(d) (6%) ***RISC-V Calling Convention***

The stack pointer is used to keep track of the memory address used as the base of the stack. The pointer value will change depending on the function calls made in the program. What is the smallest memory address which the stack pointer will reach while performing a series of **`recur_fibonacci`** function calls? Provide the calculation you made to reach your conclusion.

(e) (14%) ***RISC-V Assembly Code***

The most common way to set the base case in a recursive function is to use the conditional statement **`if`**. In **`recur_fibonacci`** function, the **`if`** statement is used to set the base case when parameter **`num`** has the value **`1`** or **`2`**.

(i) (10%) Record the Assembly code for the **`if`** statement for the base case of **`recur_fibonacci`** in Table 4 and explain the Assembly code. You may need to extend the number of rows of Table 4.

Table 4. Details of the **`if`** statement for the base case of **`recur_fibonacci`**.

Code Memory Address	Instruction	Explanation

(ii) (4%) Is it possible to optimize the Assembly code of the **`if`** statement for the base case of **`recur_fibonacci`** in terms of the number of Instruction Count? If yes, write down the optimized version of the Assembly code. If not, please justify your claim.

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

Consider a little-endian 64-bit RISC-V sequential processor. Suppose that the operating system allows for reading/writing from/into any memory, and any unsigned or signed overflow does not cause an exception.

Table 5 and 6 contain the contents in the register set, data memory, and code memory. Assume that the current program counter (**PC**) has the value **0x0000 0000 0003 00A0**.

Table 5. Initial state of register set.

Register	Initial value	Register	Initial value
x0	0x0000 0000 0000 0000	**x16**	0x0077 3670 6C4A 8054
x1	0x0000 0000 0000 F0B0	**x17**	0xE980 675D 8AEE 99C0
x2	0x0000 003E FF00 8000	**x18**	0xBA88 7751 4322 C3AD
x3	0x0000 002E 0040 0018	**x19**	0xFFFF FFA0 6000 8720
x4	0x0000 0000 003D BB30	**x20**	0xFFA0 0000 8000 A000
x5	0x0000 0000 0000 001F	**x21**	0xFFFF FFFF FFFF A100
x6	0x0000 0000 0000 003F	**x22**	0xFFFF D800 AE80 0000
x7	0x0000 002E 0040 0000	**x23**	0x0000 0000 0020 0A00
x8	0x0000 002E 0040 0038	**x24**	0x0000 0000 0200 0F10
x9	0x0000 002E 0040 0020	**x25**	0xFFFF E80E D800 A900
x10	0xA451 98CC DA45 8000	**x26**	0xFFFF 4000 3200 E000
x11	0x5022 A875 966D FF04	**x27**	0x0000 0000 0003 00A0
x12	0x0000 0001 33F0 00A0	**x28**	0x0000 0001 33F0 A000
x13	0x0000 0000 0000 00F0	**x29**	0x0000 0000 0000 0000
x14	0x0000 0000 4010 8200	**x30**	0x0000 0000 0000 000A
x15	0x033F 00AF 0000 E523	**x31**	0x0000 0000 0000 10A0

Table 6. An excerpt of the code and data memory states.

Label	Address	Initial value	Address	Initial value
	0x0000 0000 0003 0098	0x01FF 0EB3	0x0000 002E 0040 0000	0x4202 D613
	0x0000 0000 0003 009C	0xFFD1 AE23	0x0000 002E 0040 0004	0xFE06 0613
	0x0000 0000 0003 00A0	add x0, x12, x14	0x0000 002E 0040 0008	0xFD87 0713
	0x0000 0000 0003 00A4	sll x12, x12, x5	0x0000 002E 0040 000C	0x0040 006F
	0x0000 0000 0003 00A8	bge x12, x15, GE	0x0000 002E 0040 0010	0x0083 CC00
ST:	0x0000 0000 0003 00AC	jalr x1, 0(x3)	0x0000 002E 0040 0014	0x0000 0000
	0x0000 0000 0003 00B0	xor x14, x3, x14	0x0000 002E 0040 0018	0xFF91 8603
GE:	0x0000 0000 0003 00B4	jalr x1, 12(x3)	0x0000 002E 0040 001C	0x00D6 7633
	0x0000 0000 0003 00B8	srai x14, x12, 8	0x0000 002E 0040 0020	0x00C1 8F23
	0x0000 0000 0003 00BC	sra x18, x18, x5	0x0000 002E 0040 0024	0x0181 A603
	0x0000 0000 0003 00C0	lb x19, -6(x3)	0x0000 002E 0040 0028	0x00C7 4633
	0x0000 0000 0003 00C4	0xFF31 BC23	0x0000 002E 0040 002C	0x00C1 AC23
	0x0000 0000 0003 00C8	nop	0x0000 002E 0040 0030	0x4042 05B3
	0x0000 0000 0003 00CC	nop	0x0000 002E 0040 0034	0x0000 8067
	0x0000 0000 0003 00D0	nop	0x0000 002E 0040 0038	0x0000 80E7
	0x0000 0000 0003 00D4	add x0, x0, x0	0x0000 002E 0040 003C	0xFF80 80E7

(a) (22%) Trace the execution flow of the assembly code and extend Table 7 below. The execution flow stops before the instruction at the address **`0x0000 0000 0003 00C8`**.

For each executed instruction, record the following:
1) The code address.
2) The decoded instruction (if not decoded).
3) Any updated registers of new values (if any) in hexadecimal representation.
4) Any updated memory of new values (if any) in hexadecimal representation per 32-bit word.

The first two instructions have been completed for you. You may need to extend the number of rows in Table 7.

Table 7. The incomplete table of the execution flow.

Address	Decoded Instruction	Updated Register/Memory
`0x0000 0000 0003 0098`	`add x29, x30, x31`	`x29 = 0x0000 0000 0000 10AA`
`0x0000 0000 0003 009C`	`sw x29, -4(x3)`	`MEM[0x0000 002E 0040 0014] = 0x0000 10AA`
…	…	…

(b) (4%) Once you have finished the execution flow table, count the total number of memory accesses (excluding register accesses) performed throughout the execution flow. Exclude the two instructions that have been completed for you.

(c) (4%) After executing up to address **`0x0000 0000 0003 00C4`**, replace the **`nop`** instructions starting at addresses **`0x0000 0000 0003 00C8`** to perform a jump to address **`0x0000 0000 1F03 00C4`**. You can use at most three instructions.

3. (20%) ***RISC-V Assembly to C and C to RISC-V Assembly***
A C code and its equivalent RISC-V Assembly counterpart are given below. A segment of the C code is incomplete, but its RISC-V counterpart is provided. Similarly, a segment of the RISC-V Assembly code is missing, but its C code is provided. Please fill in the missing (a) C codes and (b) RISC-V Assembly instructions on the grids provided.

Some constraints are:
1) Each element of Arrays **`A`** and **`B`** is a 4-byte integer
2) The base addresses of Arrays **`A`** and **`B`** are in registers **`x5`** and **`x6`**, respectively.
3) The variables **`i`**, **`n`**, **`l`**, and **`r`** are assigned to registers **`x10`**, **`x7`**, **`x28`**, and **`x29`**, respectively.

Note: It is highly suggested that you write additional information or comments to ensure a correct answer. i.e. **`addi x5, x5, 1 # i += 1`**

```
int l = 0;
int r = 0;
int i = 0;
for (; i < n; ++i) {

// ==============================
// (a) Missing C code
// ==============================









    while (true) {
        if (i + B[i] >= n)
            break;

        if (A[B[i]] != A[i + B[i]])
            break;

        B[i]++;
    }

    if (i + B[i] > r) {
        l = i;
        r = i + B[i]
    }


}
```

```
    addi x28, x0, 0
    addi x29, x0, 0
    addi x10, x0, 0

LOOP1:
    bge  x10, x7,  END1
    bge  x10, x29, LOOP2
    slli x11, x10, 2
    add  x11,  x6, x11
    sub  x12, x29, x10
    sub  x13, x10, x28
    slli x13, x13, 2
    add  x13, x6,  x13
    lw   x14, 0(x13)
    bge  x12, x14, B1
    sw   x12, 0(x11)
    jal  x0, LOOP2
B1:
    sw   x14, 0(x11)

LOOP2:
// ==============================
// (b) Missing RISC-V Instructions
// ==============================









    addi x10, x10, 1
    jal  x0, LOOP1
END1:
```

4. (10%) ***C Functions to RISC-V Assembly***
   Implement the following C code in RISC-V Assembly. **Note**: According to RISC-V Specification, in the standard RISC-V calling convention, the stack grows downward, and the stack pointer is always kept 16-byte aligned.

```
01 long long int Func(long long int n) {
02       if (n <= 1) {
03             return 1;
04       }
05       long long int res = 0;
06       for (long long int i = 0; i < n; i++) {
07             res += Func(i) * Func(n - i - 1);
08       }
09       return res;
10 }
```