Computer Architecture HW 2

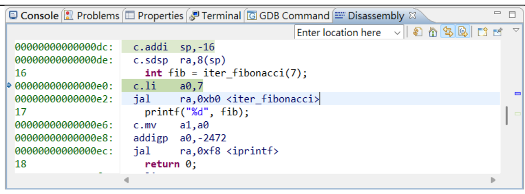
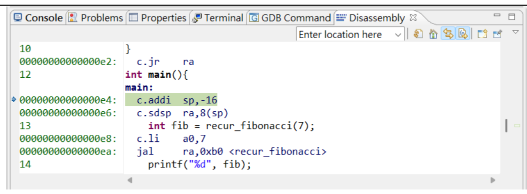
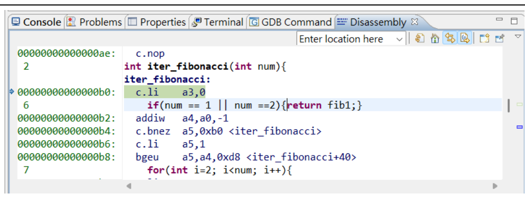
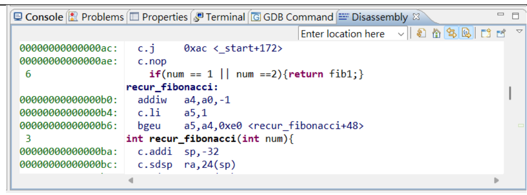
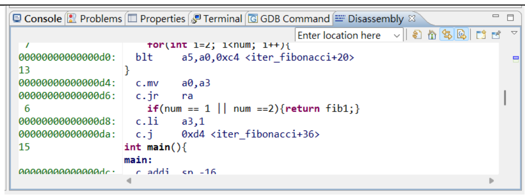
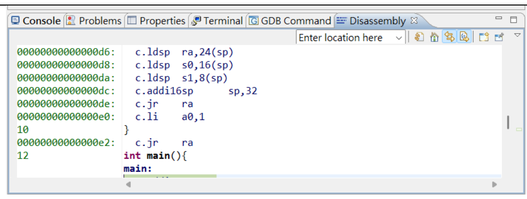
111062117, Hsiang-Sheng Huang

March 29, 2025

1

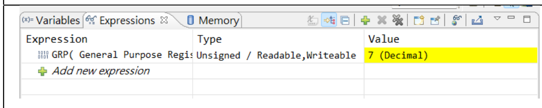
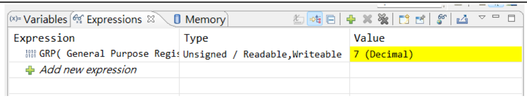
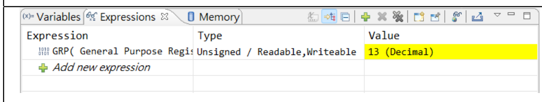
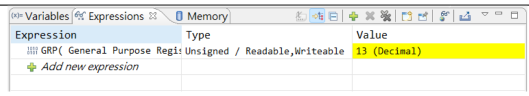
(a)

Subroutine Name	Starting Memory Address	Ending Memory Address	Reference
iter_fibonacci	0x0000 0000 0000 00b0	0x0000 0000 0000 00db	jal ra, 0xb0
recur_fibonacci	0x0000 0000 0000 00b0	0x0000 0000 0000 00e3	jal ra, 0xb0

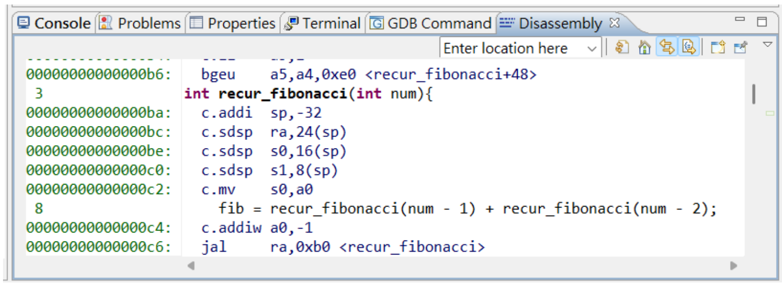
	iter	**recur**
main reference		
starting memory address		
ending memory address		

(b)

Function Name	Argument **num**		Return Value	
	Register Name	Value	Register Name	Value
iter_fibonacci	a0	7	a0	13
recur_fibonacci	a0	7	a0	13

iter	recur
	
	

(c)



Memory Address	Instruction	Saved Registers	Stack Offset
0x0000 0000 0000 00c0	c.sdsp s1,8(sp)	s1	8
0x0000 0000 0000 00be	c.sdsp s0,16(sp)	s0	16

Note that `ra` is preserved by the caller, so it is excluded from the table.

(d)

To calculate the smallest memory address reached by the stack pointer during **recur_fibonacci()** function calls, we need to analyze the recursion pattern:

$$\begin{aligned}
\texttt{recur_fibonacci}(7) &= \texttt{recur_fibonacci}(6) + \texttt{recur_fibonacci}(5) \\
&= \texttt{recur_fibonacci}(5) + \texttt{recur_fibonacci}(4) + \ldots \\
&= \texttt{recur_fibonacci}(4) + \texttt{recur_fibonacci}(3) + \ldots \\
&= \texttt{recur_fibonacci}(3) + \texttt{recur_fibonacci}(2) + \ldots \\
&= \texttt{recur_fibonacci}(2) + \texttt{recur_fibonacci}(1) + \ldots \\
&= 1 + 1 + \ldots
\end{aligned}$$

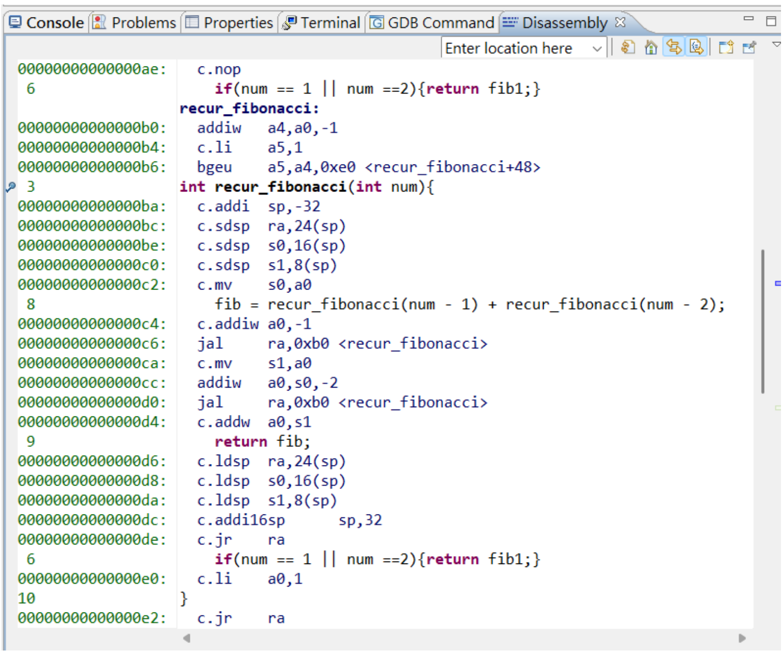
When initially entering **recur_fibonacci()**, the stack pointer value is `0x0000 0000 02FF FFF0`. With each recursive call consuming 32 bytes of stack space and a maximum recursion depth of 5 levels, the smallest memory address the stack pointer will reach is:

$$\texttt{0x0000 0000 02FF FFF0} - (32 \times 5) = \texttt{0x0000 0000 02FF FF50}$$

(e)

(i)

Code Memory Address	Instruction	Explanation
0x0000 0000 0000 00b0	`addiw a4, a0, -1`	Subtract 1 from input parameter `num` (in `a0`) and store `num-1` in `a4`. This is the first step in checking for the Fibonacci base case.
0x0000 0000 0000 00b4	`c.li a5, 1`	Load immediate value 1 into register `a5` for comparison.
0x0000 0000 0000 00b6	`bgeu a5, a4, 0xe0 <recur_fibonacci+48>`	Branch if `a5 >= a4`, meaning if `1 >= num-1`, which is equivalent to `num <= 2`. This identifies the base case where `F(1) = F(2) = 1`.
0x0000 0000 0000 00e0	`c.li a0, 1`	Load immediate value 1 into register `a0` for the return value of the base case.
0x0000 0000 0000 00e2	`c.jr ra`	Jump to the return address stored in `ra`.



(ii)

Yes, it is possible to optimize the Assembly code for the base case check in **recur_fibonacci()**. Optimized implementation:

```
c.li a5, 3                              # Load immediate 3 into a5
bltu a0, a5, 0xe0 <recur_fibonacci+48>  # Branch if num < 3
c.li a0, 1                              # Load immediate 1 into a0 (return value)
c.jr ra                                 # Jump to return address
```

This optimization eliminates the subtraction operation and directly compares the input parameter.

2

(a)

Address	texttt 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
0x0000 0000 0003 00A0	add x0, x12, x14	Nothing (x0 is immutable)
0x0000 0000 0003 00A4	sll x12, x12, x5	x12 ← x12 « 31 = 0x99F8 0050 0000 0000
0x0000 0000 0003 00A8	bge x12, x15, GE	(Nothing)
0x0000 0000 0003 00AC	jalr x1, 0(x3)	x1 ← PC + 4 = 0x0000 0000 0003 00B0
0x0000 002E 0040 0018	lb x12, -7(x3)	x12 ← 0xFFFF FFFF FFFF FFCC (sign-extension)
0x0000 002E 0040 001C	and x12, x12, x13	x12 ← 0x0000 0000 0000 00C0
0x0000 002E 0040 0020	sb x12, 30(x3)	MEM[0x0000 002E 0040 0034] = 0x00C0 8067
0x0000 002E 0040 0024	lw x12, 24(x3)	x12 ← 0x0000 0000 4042 05B3
0x0000 002E 0040 0028	xor x12, x14, x12	x12 ← 0x0000 0000 0052 87B3
0x0000 002E 0040 002C	sw x12, 24(x3)	MEM[0x0000 002E 0040 0030] = 0x0052 87B3
0x0000 002E 0040 0030	add x15, x5, x5	x15 ← 0x0000 0000 0000 003E
0x0000 002E 0040 0034	jalr x0, 12(x1)	(Nothing)
0x0000 0000 0003 00BC	sra x18, x18, x5	x18 ← 0xFFFF FFFF 7510 EEA2
0x0000 0000 0003 00C0	lb x19, -6(x3)	x19 ← 0xFFFF FFFF FFFF FF83
0x0000 0000 0003 00C4	sd x19, -8(x3)	MEM[0x0000 002E 0040 0010] = 0xFFFF FF83 MEM[0x0000 002E 0040 0014] = 0xFFFF FFFF

(b)

Numbers of memory accesses:

- Instruction fetches: 15
- Load operations: 3
- Store operations: 3

So the total number of memory accesses is $15 + 3 + 3 = 21$.

(c)

We notice that the target address 0x0000 0000 1F03 00C4 can be expressed as:

$$\texttt{0x1F03 00C4} = \texttt{0x0003 00C8} + \texttt{0x1F00 0000} - \texttt{0x0000 0004}$$

So we can use auipc first to load the upper 20 bits of the address, and then use jalr to jump to the target address.

```
auipc x5, 0x1F000     # x5 ← PC + 0x1F000000
jalr x0, -4(x5)       # Jump to x5 - 4 (target: 0x0000 0000 1F03 00C4)
```

3

The overall code implements the Z algorithm for pattern matching. The missing C code contains the logic for updating array B based on previously computed values. The missing RISC-V instruction segment contains the implementation that updates arrays A and B within the main computation loop of the algorithm.

(a)

The missing C code is:

```c
int l = 0;
int r = 0;
int i = 0;
for (; i < n; ++i) {
    // missing C code
    if (i < r) {
        int k = i - l;
        if (r - i < B[k]) {
            B[i] = r - i;
        } else {
            B[i] = B[k];
        }
    }
    // end of missing C code
}
```

(b)

The missing RISC-V Instruction is:

```
LOOP2:
# --------------- missing RISC-V instruction ---------------
    lw x15, 0(x11)      # x15 ← B[i]
    add x16, x10, x15   # x16 ← i + B[i]
    bge x16, x7, B2     # if (i + B[i] >= n) goto B2
    slli x17, x15, 2    # x17 ← B[i] << 2
    add x17, x15, x17   # x17 ← &A[B[i]]
    lw x17, 0(x17)      # x17 ← A[B[i]]
    slli x18, x16, 2    # x18 ← (i + B[i]) << 2
    add x18, x5, x18    # x18 ← &A[i + B[i]]
    lw x18, 0(x18)      # x18 ← A[i + B[i]]
    bne x17, x18, B2    # if (A[B[i]] != A[i + B[i]]) goto B2
    addi x15, x15, 1    # B[i]++
    sw x15, 0(x11)      # B[i] ← x15 (store back to memory)
    jal x0, LOOP2       # goto LOOP2
B2:
    bge x29, x16, B3    # if (i + B[i] <= r) goto B3
    addi x28, x10, 0    # l ← i
    addi x29, x16, 0    # r ← i + B[i]
B3:
# --------------- end of missing RISC-V instruction ---------------
    addi x10, x10, 1    # i++
    jal x0, LOOP1       # goto LOOP1
```

4

```
Func:
    # Allocate stack frame for ra, s0, s1, s2 (total 32 bytes)
    addi x2, x2, -32        # sp ← sp - 32
    sd   x1, 24(x2)         # save return address
    sd   x8, 16(x2)         # save s0
    sd   x9, 8(x2)          # save s1
    sd   x18, 0(x2)         # save s2

    add  x8, x0, x10        # x8 ← x10 (save n to s0)

    # Base case: if (n <= 1) return 1
    addi x5, x0, 1          # x5 ← 1
    blt  x5, x8, L0         # if n > 1, goto L0
    addi x10, x0, 1         # x10 ← 1 (return value)
    jalr x0, 0(x1)          # return

L0:
    addi x9, x0, 0          # x9 ← i = 0
    addi x18, x0, 0         # x18 ← res = 0

LOOP:
    bge  x9, x8, DONE       # if i >= n, goto DONE

    # Call Func(i)
    add  x10, x0, x9        # argument = i
    jal  x1, Func
    add  x6, x0, x10        # x6 ← Func(i)

    # Call Func(n - i - 1)
    sub  x7, x8, x9         # x7 ← n - i
    addi x7, x7, -1         # x7 ← n - i - 1
    add  x10, x0, x7        # argument = n - i - 1
    jal  x1, Func
    mul  x6, x6, x10        # x6 ← Func(i) * Func(n - i - 1)
    add  x18, x18, x6       # res += Func(i) * Func(n - i - 1)

    addi x9, x9, 1          # i++
    beq  x0, x0, LOOP       # unconditional jump to LOOP

DONE:
    add  x10, x0, x18       # return res

    # Restore ra, s0, s1, s2
    ld   x18, 0(x2)         # restore s2
    ld   x9,  8(x2)         # restore s1
    ld   x8, 16(x2)         # restore s0
    ld   x1, 24(x2)         # restore return address
    addi x2, x2, 32         # deallocate stack
    jalr x0, 0(x1)          # return
```