Xi'an Jiaotong-Liverpool University

西交利物浦大学

CPT210 - Microprocessor

Lecture 7 – ARM Functions and Stacks-Part 2

Overview

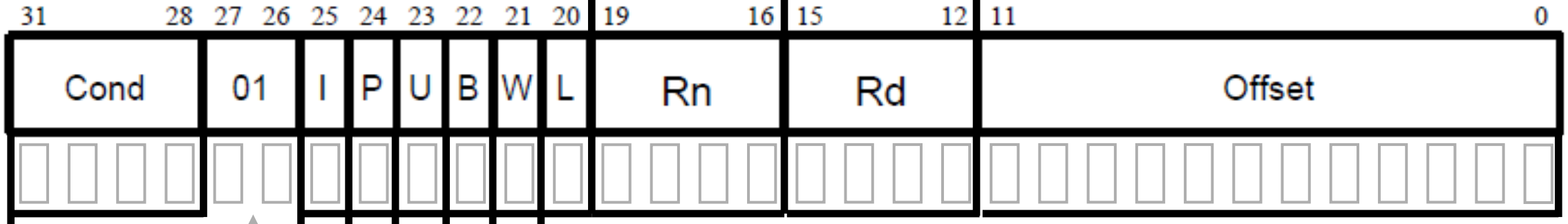
- Lecture 6 Review
  - LDR/STR
  - ADR
  - LDR Pseudo-instruction
- Lecture 7
  - Functions
  - Stacks
  - A Case Study

Before End: AY2024-25 Semester 2 End-of-Semester Module Questionnaire

Before End: Assessments' Instructions & Structure

Review of Lecture 6

- *LDR/STR Instruction Encoding*



Base register: Register holding the base memory address

Destination/source register: Where data is stored or loaded from

Offset: 12-bit offset or shifted register value (depends on **I**)

Instruction class: Always **01** for Load/Store instructions

**different from data processing instructions

Load/Store: **1**=Load (LDR); **0**=Store (STR)

Write-back bit: **1**=Write result back to base register transfer; **0**=Don't write back

Byte/Word bit: **1**=Byte transfer; **0**=Word transfer

Up/Down bit: **1**=Add offset to base; **0**=Subtract offset from base

Pre/Post indexing: **1**=Pre-indexing (add offset before transfer; **0**=Post-indexing (after)

Immediate bit: **0**=**offset** is a 12-bit immediate value; **1**=**offset** is a register

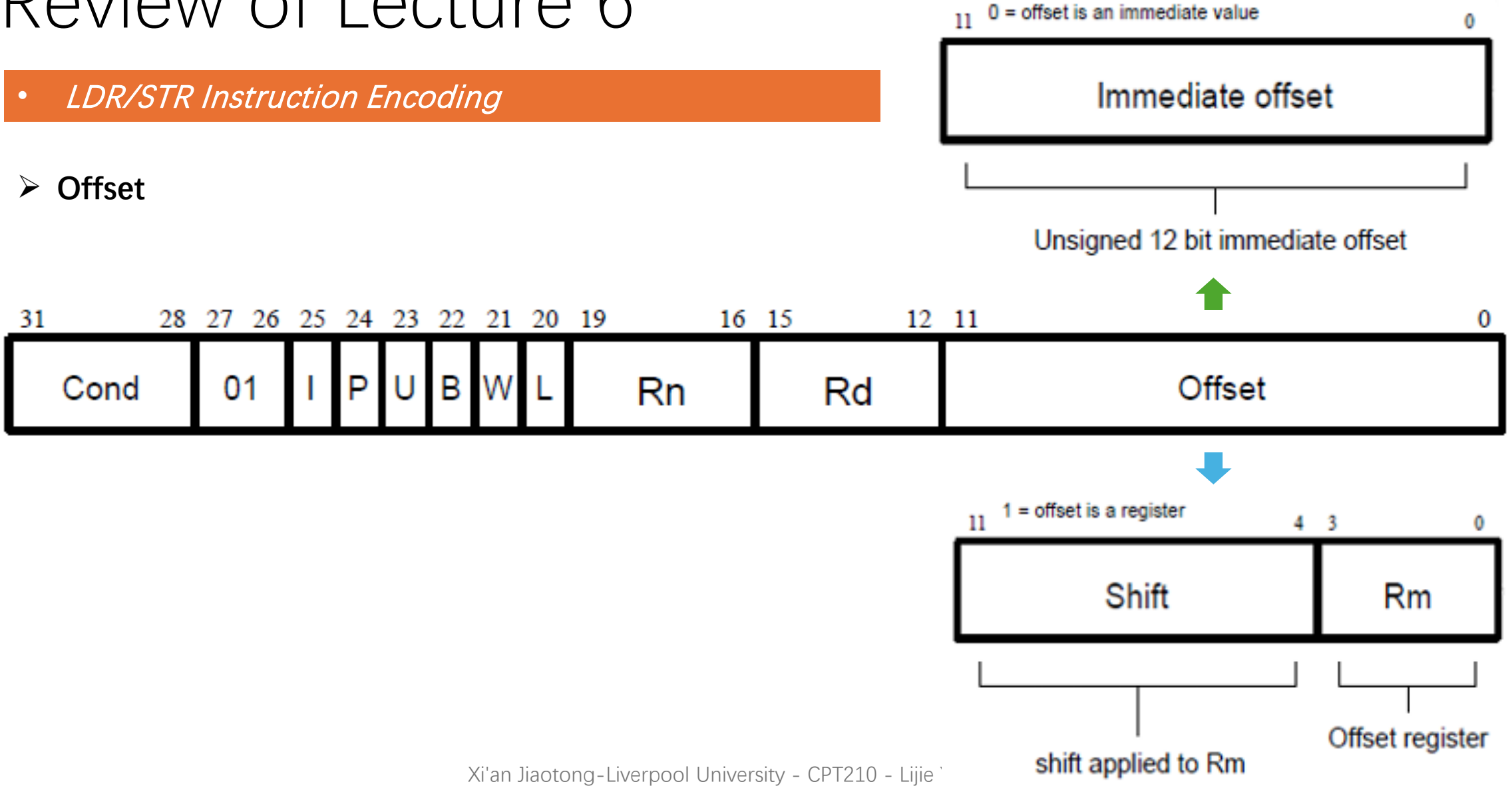
**opposite of the data processing instructions

Condition field: Executes only if condition is met

Review of Lecture 6

- *LDR/STR Instruction Encoding*

➢ **Offset**

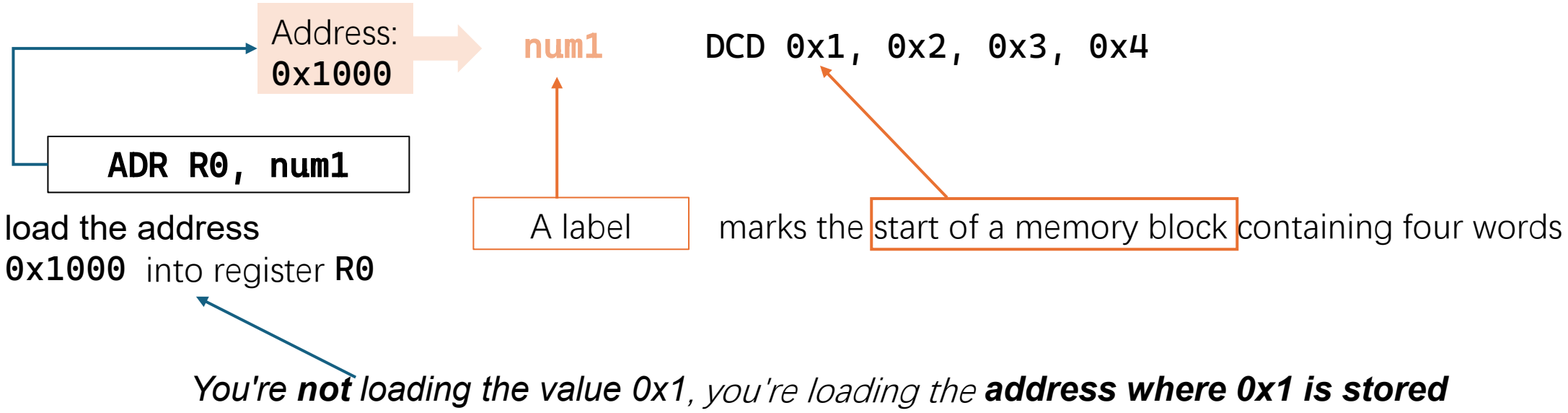


Review of Lecture 6

- *ADR*

➢ **What is ADR?**

ADR stands for **Address Register**.
It is an instruction to load the **address of a label** (memory location) into a register.



Review of Lecture 6

- *LDR Pseudo-Instruction*

```
LDR R0, =0xAABBCCDD
```

pseudo-instruction

🔧 **What does the assembler actually do?**

1. It places the constant (0xAABBCCDD) somewhere **near your instruction**, in a region called the **literal pool**
2. It then converts the pseudo-instruction into a real instruction like:

```
LDR R0, [PC, #offset]
```

This loads the value from **memory near the PC**, not directly from the instruction.

Lecture 7

Functions

Implementing functions that support multi-level calls

Function Calls in ARM

- Elements of a function: function name, return value, parameters, local variables, function logic.
- ARM's branch and link instruction, `BL`, automatically saves the return address (the next instruction's address) in the register R14 (i.e. LR).
- We can use "`MOV PC, LR`" at the end of the subroutine to return back to the instruction after the subroutine call `BL SUBROUTINE_NAME`.
  - A `SUBROUTINE_NAME` is a label in the ARM program.
- However, BL and BX/MOV alone are not enough to support function calls:

Any Problems?

example 4.2

```
1  main
2              mov         r0, #1
3              mov         r1, #3
4              mov         r2, #6
5              bl          sum         ; call f
6              end         ; end of main
7
8              ;           f assumes parameters in r0, r1, r2
9              ;           and saves r0 + r1 + (2 * r2) to r9
10 sum
11             add         r5, r0, r1
12             mov         r0, r2      ; prepare to call g
13             bl          double
14             add         r9, r9, r5
15             mov         pc, lr      ; return
16
17             ;           g assumes one parameter in r0
18             ;           and saves 2 *r0 to r9
19 double
20             add         r9, r0, r0
21             mov         pc, lr      ; return
```

end stops the program

Example 4.2: Discussion

- Execution flow: main -> sum -> double -> sum -> ???
  - Run this program to find out

- The instructions `bl` and "`mov pc, lr`" do not work if a called function calls another function.
  - The register `LR` will be overwritten at the second function call.
  - The first value of `LR` will be lost.

- Workaround: backup LR to another register before calling a function and restore LR once that function returns:

example 4.3

```
1  main
2          mov     r0, #1
3          mov     r1, #3
4          mov     r2, #6
5          bl      sum         ; call f
6          end     ; end of main
7
8          ;       f assumes parameters in r0, r1, r2
9          ;       and saves r0 + r1 + (2 * r2) to r9
10 sum
11         mov     r8, lr      ; backup lr for the calling function (main)
12         add     r5, r0, r1
13         mov     r0, r2      ; prepare to call g
14         bl      double
15         add     r9, r9, r5
16         mov     lr, r8      ; restore lr for the calling function
17         mov     pc, lr      ; return
18
19         ;       g assumes one parameter in r0
20         ;       and saves 2 *r0 to r9
21 double
22         add     r9, r0, r0
23         mov     pc, lr      ; return
```

Example 4.3: Discussion

- Execution flow: main -> sum -> double -> sum -> main
  - Works properly now.
- But what if the function "double" calls another function?
  - Need another register to backup LR.
- How many registers do we have? Enough?
  - Need registers to backup LR
  - Need registers to backup parameters
  - Need registers to backup local variables of each function.
  - Registers will soon be all used.
- Need to make use of memory!
  - 16 GB vs a few 32-bit registers

ARM Function Call

- To support multi-level function calls (also called **nested subroutines**), we need to use memory to store local variables, parameters, return addresses.
  - The memory usage grows when a function is called.
  - The memory usage shrinks when a function returns.
  - We use a data structure called **stack** to achieve this.
- Parameters are passed to the function through r0 ~ r3.
  - What if r0 ~ r3 are not enough?

- SP (r13) records the current position of the stack.
- FP (r11) is the frame pointer (not always used).
- LR (r14) records the return address.
- PC (r15) is the program counter.

Lecture 7

```
BL my_function   ; Save return address in LR, jump
BX LR            ; Jump back to caller, maybe switch mode
```

- *Differences between BL and BX*

Feature	**BL (Branch with Link)**	**BX (Branch and Exchange)**
Purpose	Save return address and **Call a subroutine**	**Branch (jump)** to a (register) address, (optionally) **switch modes**
Saves return address?	✅ Yes, into **LR (R14)**	❌ No, does **not save** any return address
Used for?	Calling functions/subroutines	Returning from functions, mode switching
Instruction format	BL LABEL	BX REGISTER (e.g., BX LR)
Mode switching	❌ No mode switching	✅ Switches between **ARM** and **Thumb** modes
Return from subroutine?	❌ Used to **enter** a subroutine	✅ Often used to **return** from a subroutine

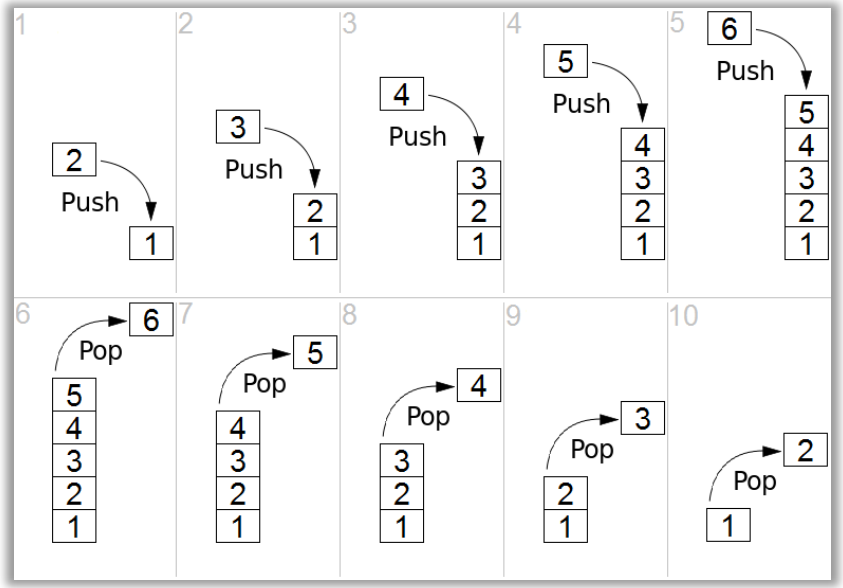
Lecture 7

Stack

Implementing functions that support multi-level calls

The Stack

- Stack is a data structure, known as **last in first out (LIFO).**
- In a stack, the **last** item you **put in** is the **first** item you **take out**.
- Stacks in microprocessors are implemented by using a stack pointer to point to the top of the stack in memory.
  - As items are added to the stack (pushed), the stack pointer is moving up
  - and as items are removed from the stack (pulled or popped), the stack pointer is moved down.



In ARM, a stack is used to:

- Save local variables
- Save return addresses when calling functions
- Support nested function calls

ARM uses a **Stack Pointer (SP)**, usually **R13**, to keep track of the **top of the stack**.

Stack Classification

- Based on the **direction of stack growth**:
  - Ascending Stack - When items are pushed on to the stack, the stack pointer is increasing. That means the stack grows towards higher address.
  - Descending Stack - When items are pushed on to the stack, the stack pointer is decreasing. That means the stack is growing towards lower address.

- Based on **where the stack pointer points to**:
  - Empty Stack - Stack pointer points to the location in which the next/first item will be stored. e.g. A push will store the value, and increment the stack pointer for an Ascending Stack.
  - Full Stack - Stack pointer points to the location in which the last item was stored. e.g. A pop will decrement the stack pointer and pull the value for an Ascending Stack.

Stack Operation Instructions

- There are two instructions in VisUAL that support partial stack operations: STM and LDM

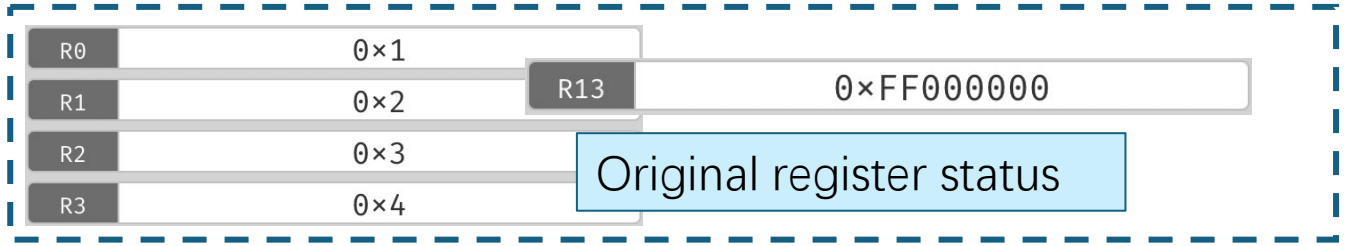
Often R13 (i.e. SP)

Syntax	Means
`STM{addr_mode}{cond} Rn{!}, reglist`	**Store** multiple registers to memory
`LDM{addr_mode}{cond} Rn{!}, reglist`	**Load** multiple registers from memory

- addr_mode can be:
  - IA: Increment address After each transfer
  - IB: Increment address Before each transfer
  - DA: Decrement address After each transfer
  - DB: Decrement address Before each transfer.
- {!}: if ! is present, the final address is written back into *Rn*.
- reglist: is a list of one or more registers to be loaded/saved

example 4.4

```
mov     r0, #1
mov     r1, #2
mov     r2, #3
stmdb   r13!, {r0, r1, r2}
ldmia   sp!, {r3, r4, r5}
```



✅ **Key Rule:**

ARM automatically sorts the register list internally based on register number, from **lowest** to **highest, regardless of the order you write them.**

stmia sp!, {r0, r1, r2, r3}

Empty Ascending Stack

Address	Value
0×FF000000	0×1
0×FF000004	0×2
0×FF000008	0×3
0×FF00000C	0×4

R13 0×FF000010

stmib sp!, {r0, r1, r2, r3}

Full Ascending Stack

Address	Value
0×FF000004	0×1
0×FF000008	0×2
0×FF00000C	0×3
0×FF000010	0×4

R13 0×FF000010

stmda sp!, {r0, r1, r2, r3}

Empty Descending Stack

Address	Value
0×FEFFFFF4	0×1
0×FEFFFFF8	0×2
0×FEFFFFFC	0×3
0×FF000000	0×4

R13 0×FEFFFFF0

stmdb sp!, {r0, r1, r2, r3}

Full Descending Stack

Address	Value
0×FEFFFFF0	0×1
0×FEFFFFF4	0×2
0×FEFFFFF8	0×3
0×FEFFFFFC	0×4

R13 0×FEFFFFF0

Stack Operation Instructions

- If a full descending stack is used, you can simple use the instructions: PUSH and POP (NOT supported by VisUAL)

Syntax	Example
POP{*cond*} *reglist*	pop {r7, pc}
PUSH{*cond*} *reglist*	push {r7, lr}

POP/PUSH can be understood as pseudo-instructions of LDMIA/STMDB

- `POP` has the same effect as "`LDMIA sp!, reglist`".
- `PUSH` has the same effect as "`STMDB sp!, reglist`".

```c
int a = 12;
int b = 15;

int f(int n1, int n2)
{
        a = a + n1;
        b = b + n2;
        return a + b;
}

int g(int n1, int n2, int n3, int n4, int n5, int n6)
{
        return n1 + n2 + n3 + n4 + n5 + n6;
}

int main()
{
        int x = f(1, 2);
        int y = g(1, 2, 3, 4, 5, 6);
        return 0;
}
```

Stack in Action (Example 4.5 C)

```java
public class example_4_5 {
    public static int a = 12;
    public static int b = 15;

    public static int f(int n1, int n2)
    {
        a = a + n1;
        b = b + n2;
        return a + b;
    }

    public static int g(int n1, int n2, int n3,
                        int n4, int n5, int n6)
    {
        return n1 + n2 + n3 + n4 + n5 + n6;
    }

    public static void main(String[] args)
    {
        int x = f(1, 2);
        int y = g(1, 2, 3, 4, 5, 6);
    }
}
```

Stack in Action (Example 4.5 Java)

Example 4.5: Discussion

Look at the C or Java code and compare them against their corresponding assembly code. Answer questions below:

- Question 1: How are the parameters of function g passed from main?
- Question 2: How are the return values of f and g passed to main?
- Question 3: Can you map every line of the function f to its assembly code?
- Question 4: Where are the local variables of f and g stored?
- Question 5: Where are variables a and b stored?

Example 4.5: Answers

Example 4_5_explained.s

Question 1: How are the parameters of function **g** passed from main?

ARM calling convention for **first 4 parameters**: passed in **R0, R1, R2, R3**.
For **g(int n1, n2, n3, n4, n5, n6)**:

- **n1 → R0**
- **n2 → R1**
- **n3 → R2**
- **n4 → R3**
- **n5** and **n6** are passed **on the stack**:
  - **n5 → [SP, #offset]**
  - **n6 → [SP, #offset+4]**

Example 4.5: Answers

Example 4_5_explained.s

- Question 2: How are the return values of **f** and **g** passed to main?

  - ARM returns values using **R0**.
  - After calling **f()** or **g()**, the result is in **R0**, and then it's moved or stored to the local variable (**x** or **y**) in **main**.

```
f:
    @ Function supports interworking.
    @ args = 0, pretend = 0, frame = 8
    @ frame_needed = 0, uses_anonymous_args = 0
    @ link register save eliminated.
    sub sp, sp, #8
    str r0, [sp, #4]
    str r1, [sp]
    ldr r3, .L3
    ldr r2, [r3]
    ldr r3, [sp, #4]
    add r3, r2, r3
    ldr r2, .L3
    str r3, [r2]
    ldr r3, .L3+4
    ldr r2, [r3]
    ldr r3, [sp]
    add r3, r2, r3
    ldr r2, .L3+4
    str r3, [r2]
    ldr r3, .L3
    ldr r2, [r3]
    ldr r3, .L3+4
    ldr r3, [r3]
    add r3, r2, r3
    mov r0, r3
    add sp, sp, #8
    @ sp needed
    bx  lr
```

```
g:
    @ Function supports interworking.
    @ args = 8, pretend = 0, frame = 16
    @ frame_needed = 0, uses_anonymous_args = 0
    @ link register save eliminated.
    sub sp, sp, #16
    str r0, [sp, #12]
    str r1, [sp, #8]
    str r2, [sp, #4]
    str r3, [sp]
    ldr r2, [sp, #12]
    ldr r3, [sp, #8]
    add r2, r2, r3
    ldr r3, [sp, #4]
    add r2, r2, r3
    ldr r3, [sp]
    add r2, r2, r3
    ldr r3, [sp, #16]
    add r2, r2, r3
    ldr r3, [sp, #20]
    add r3, r2, r3
    mov r0, r3
    add sp, sp, #16
    @ sp needed
    bx  lr
```

Example 4.5: Answers

Example 4_5_explained.s

```c
int f(int n1, int n2)
{
        a = a + n1;
        b = b + n2;
        return a + b;
}
```

- Question 3: Can you map every line of the function **f** to its assembly code?

```
f:
    @ Function supports interworking.
    @ args = 0, pretend = 0, frame = 8
    @ frame_needed = 0, uses_anonymous_args = 0
    @ link register save eliminated.
    sub sp, sp, #8
    str r0, [sp, #4]
    str r1, [sp]
    ldr r3, .L3
    ldr r2, [r3]
    ldr r3, [sp, #4]
    add r3, r2, r3
    ldr r2, .L3
    str r3, [r2]
    ldr r3, .L3+4
    ldr r2, [r3]
    ldr r3, [sp]
    add r3, r2, r3
    ldr r2, .L3+4
    str r3, [r2]
    ldr r3, .L3
    ldr r2, [r3]
    ldr r3, .L3+4
    ldr r3, [r3]
    add r3, r2, r3
    mov r0, r3
    add sp, sp, #8
    @ sp needed
    bx  lr
```

```
sub sp, sp, #8      ; Allocate 8 bytes stack space
str r0, [sp, #4]    ; Save n1 at sp+4
str r1, [sp]        ; Save n2 at sp
```

C Equivalent:
n1 and **n2** (parameters) saved on stack for use later.

```
ldr r3, .L3         ; Load address of global variable a
ldr r2, [r3]        ; Load value of a → r2
ldr r3, [sp, #4]    ; Load n1 from stack → r3
add r3, r2, r3      ; a + n1 → r3
ldr r2, .L3         ; Load address of a again
str r3, [r2]        ; Store new a = a + n1
```

C Equivalent:
•Load **a**, add **n1**, and store back to **a**.

Example 4.5: Answers

Example 4_5_explained.s

```c
int f(int n1, int n2)
{
        a = a + n1;
        b = b + n2;
        return a + b;
}
```

- Question 3: Can you map every line of the function **f** to its assembly code?

```
f:
    @ Function supports interworking.
    @ args = 0, pretend = 0, frame = 8
    @ frame_needed = 0, uses_anonymous_args = 0
    @ link register save eliminated.
    sub sp, sp, #8
    str r0, [sp, #4]
    str r1, [sp]
    ldr r3, .L3
    ldr r2, [r3]
    ldr r3, [sp, #4]
    add r3, r2, r3
    ldr r2, .L3
    str r3, [r2]
    ldr r3, .L3+4
    ldr r2, [r3]
    ldr r3, [sp]
    add r3, r2, r3
    ldr r2, .L3+4
    str r3, [r2]
    ldr r3, .L3
    ldr r2, [r3]
    ldr r3, .L3+4
    ldr r3, [r3]
    add r3, r2, r3
    mov r0, r3
    add sp, sp, #8
    @ sp needed
    bx  lr
```

```
ldr r3, .L3+4       ; Load address of b
ldr r2, [r3]        ; Load value of b → r2
ldr r3, [sp]        ; Load n2 from stack → r3
add r3, r2, r3      ; b + n2 → r3
ldr r2, .L3+4       ; Load address of b again
str r3, [r2]        ; Store new b = b + n2
```

C Equivalent:
- Load **b**, add **n2**, and store back to **b**.

```
ldr r3, .L3         ; Load address of a
ldr r2, [r3]        ; Load updated value of a → r2
ldr r3, .L3+4       ; Load address of b
ldr r3, [r3]        ; Load updated value of b → r3
add r3, r2, r3      ; a + b → r3
mov r0, r3          ; Return value in r0
```

C Equivalent:
- Return **a + b**.
- **r0** holds return value (standard ARM return convention).

Example 4.5: Answers

Example 4_5_explained.s

```c
int f(int n1, int n2)
{
        a = a + n1;
        b = b + n2;
        return a + b;
}
```

- Question 3: Can you map every line of the function **f** to its assembly code?

```
f:
    @ Function supports interworking.
    @ args = 0, pretend = 0, frame = 8
    @ frame_needed = 0, uses_anonymous_args = 0
    @ link register save eliminated.
    sub sp, sp, #8
    str r0, [sp, #4]
    str r1, [sp]
    ldr r3, .L3
    ldr r2, [r3]
    ldr r3, [sp, #4]
    add r3, r2, r3
    ldr r2, .L3
    str r3, [r2]
    ldr r3, .L3+4
    ldr r2, [r3]
    ldr r3, [sp]
    add r3, r2, r3
    ldr r2, .L3+4
    str r3, [r2]
    ldr r3, .L3
    ldr r2, [r3]
    ldr r3, .L3+4
    ldr r3, [r3]
    add r3, r2, r3
    mov r0, r3
    add sp, sp, #8
    @ sp needed
    bx  lr
```

```
add sp, sp, #8        ; Restore stack pointer
bx lr                 ; Return to caller
```

Epilogue:
Restore stack and return

Example 4.5: Answers

Example 4_5_explained.s

- Question 4: Where are the local variables of **f** and **g** stored?

  - Local variables in **f** and **g** (if any) are stored on the **stack (SP)**.
  - Parameters **n1, n2** are in **R0, R1**.
  - Since **f** and **g** don't define additional local variables besides parameters, **only stack is used** for:
    - Saving **LR**, maybe **FP**, or extra arguments beyond **R0–R3**.

Example 4.5: Answers

Example 4_5_explained.s

- Question 5: Where are variables **a** and **b** stored?

  - **a** and **b** are global variables.
  - They are stored in the **data section** of memory
  - Accessed via **LDR/STR** instructions

Important Things about Functions

- Each function has a stack frame, which holds all local variables (including parameters) of that function
  - What is the stack frame of the function f?
- Global variables are not stored inside stack.
  - The ".word" you see in the example is similar to "DCD" we used before. They are followed by the values of these global variables.
- Some more information: https://www.techopedia.com/definition/22304/stack-frame#:~:text=A%20stack%20frame%20is%20comprised%20of%3A%201%20Local,need%20restoration%203%20Argument%20parameters%204%20Return%20address

Lecture 7

A Case Study

Recursive function calls in assembly and tail call elimination in action

Concept 1: Recursive Function Calls

- Recursive function: A function "that calls itself one or more times until a specified condition is met at which time the rest of each repetition is processed from the last one called to the first".

- Below is an example of a recursive function (example 4.6):

```
int f(int y, int sum)
{
        if (y == 0) {
                return sum;
        } else {
                sum = sum + y;
                return f(y - 1, sum);
        }
}
```

Concept 2: Tail Call and Tail-Recursive

- A **tail call** is a function call performed as the final action of a procedure.

```
int foo(data) {
    a(data); // No
    return b(data); // Yes
}
```

```
int bar(data) {
    if ( a(data) ) { // No
        return b(data); // Yes
    }
    return c(data); // Yes
}
```

- If a tail call might lead to the same function being called again later in the call chain, the function is called **tail-recursive**.
  - Example 4.6 is tail-recursive

Example 4.6

```c
#include <stdio.h>

int f(int y, int sum)
{
    if (y == 0) {
        return sum;
    } else {
        sum = sum + y;
        return f(y - 1, sum);
    }
}

int main(void)
{
    int x;
    scanf("%d", &x);
    x= f(x, 0);
    printf("%d\n", x);
    return x;
}
```

Tail Call: More Examples

- Do the following examples involve tail calls?

```
int foo1(data) {
    return a(data) + 1;
}
```

```
int foo2(data) {
    int ret = a(data);
    return ret;
}
```

Tail Call: More Examples

- Do the following examples involve tail calls?

```
int foo1(data) {
    return a(data) + 1;
}
```

No!

Why?

- After **a(data)** returns, **+ 1** still needs to be added.
- This means further computation is required after the function call.
- So this cannot be optimized as a tail call.

```
int foo2(data) {
    int ret = a(data);
    return ret;
}
```

Tail Call: More Examples

- Do the following examples involve tail calls?

```
int foo1(data) {
    return a(data) + 1;
}
```

```
int foo2(data) {
    int ret = a(data);
    return ret;
}
```

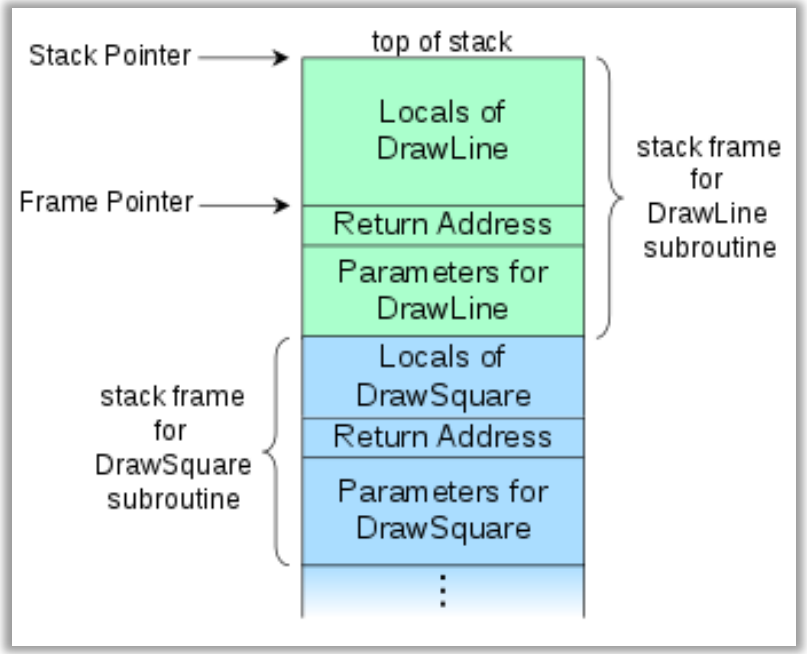
YES

Why?

- The result of **a(data)** is directly returned without modification.
- No further computation after the call.
- This can be optimized as a tail call.

Tail Call Optimisation

- If you let a function call itself for many times, it will eventually consume too much memory that the operating system will kill the program.
- For tail calls, they can be implemented without adding a new stack frame to the **call stack**.
- This optimisation technique is called tail call elimination.
  - Tail recursions can be as efficient as `for` loops



Tail Call Optimisation

Normal Function Calls:
- Each function call creates a new stack frame.
- Deep recursion can quickly consume stack space → Stack Overflow.

Tail Calls:
- In a Tail Call, the function doesn't need to keep its own stack frame.
- The current stack frame can be reused for the next function call.
- This makes recursion as efficient as loops — no extra memory needed!

🧠 **In Short:**
Tail calls helps you write faster, safer, and smarter recursive code, and gives you insight into how real programs run at a low level.

Example 4.6 – Try yourselves

- The example 4.6 has two pieces of associated assembly code.
  - One is compiled normally.
  - Another has tail call optimisation on.

- Question 1: Can you map every line of the function to its assembly (both versions)?

- Question 2: Why it is safe without adding a new stack frame to the call stack?

Before End:
AY2024-25 Semester 2 End-of-Semester Module Questionnaire
(introduce as required by the university)

Step 1:

Open a browser, type the link and enter: https://mymq.xjtlu.edu.cn/

Or, scan the QR code below:

Step 2:

Find **CPT210 – Microprocessor System** in University-Level Module Questionnaire

Welcome to the XJTLU Module Questionnaire (MQ) System

The **XJTLU Module Questionnaire (MQ)** provides students with a formal platform to confidentially share their perspectives on modules and teaching quality.

The MQ system operates on two distinct levels to ensure comprehensive feedback collection:

1. University-Level Module Questionnaire
Administered centrally by the Registry, this semester-based survey gathers feedback through a standard questionnaire at the end of the module teaching.

2. School/Academy–Level Module Questionnaire
Individual academic units conduct tailored surveys aligned with their disciplinary needs during the semester.

All responses remain strictly confidential and are processed anonymously. Insights from the MQ directly contribute to data-driven decision-making, enabling the University and academic units to address student needs and uphold educational excellence.

Contact Us
For inquiries or support regarding the MQ system, please refer to the ***MQ User Guide*** or reach out to the MQ team at **MQ@xjtlu.edu.cn**.

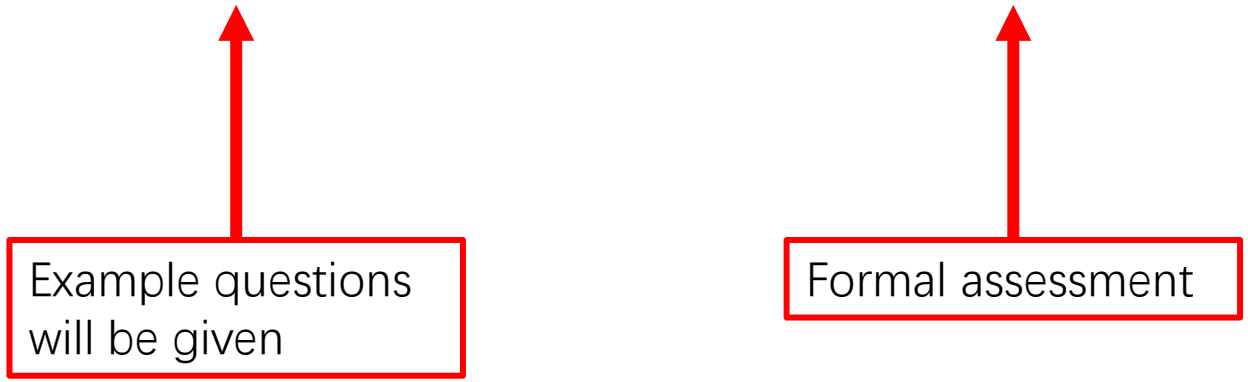
Step 3:

Start your questionnaire, complete and submit it.

Before End:
Assessments' Instructions and Structure

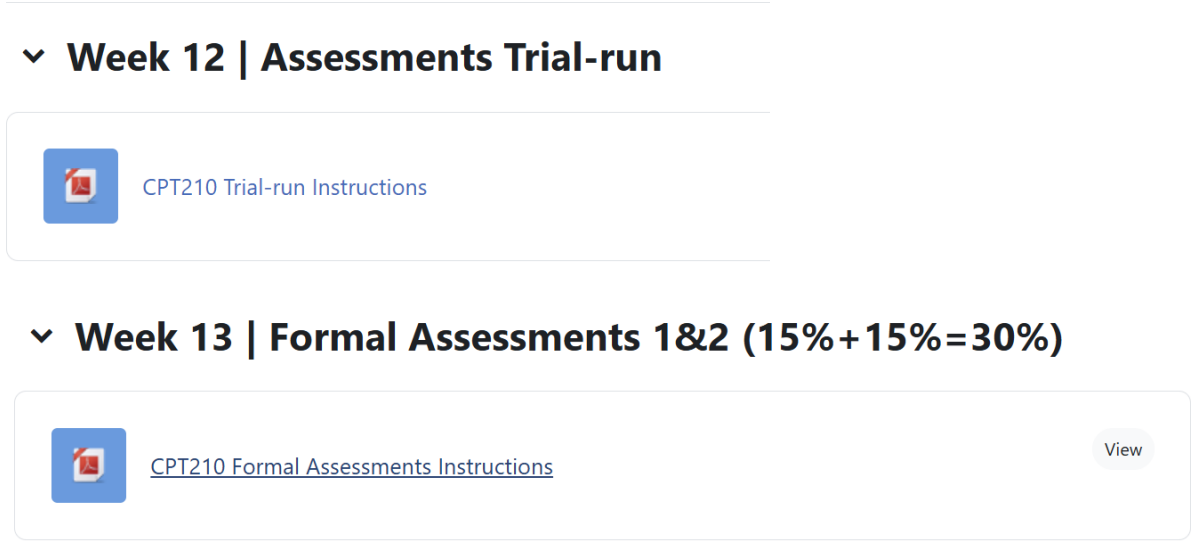
Things to prepare and bring:

1. Your **fully-charged** personal **LAPTOPs**, **not tablets, not smartphones**
2. Well connect your **laptops** to campus **WIFI**
3. Show up **in-person** on Tutorial on **May 9th** for a trial-run and **May 16th** for the formal assessment



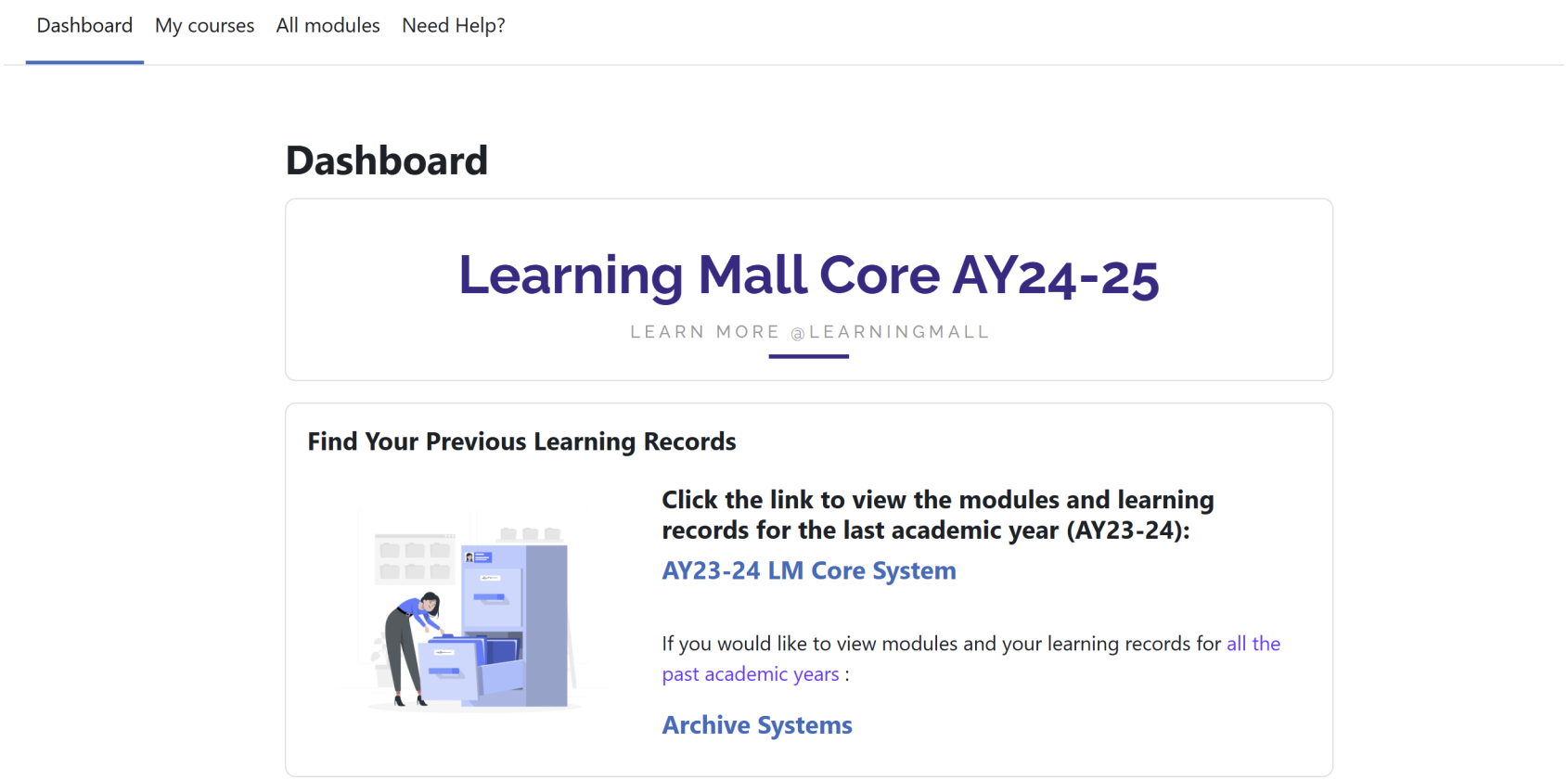
Instructions:

Instruction handbooks are available on LMO, CPT210 Module page:



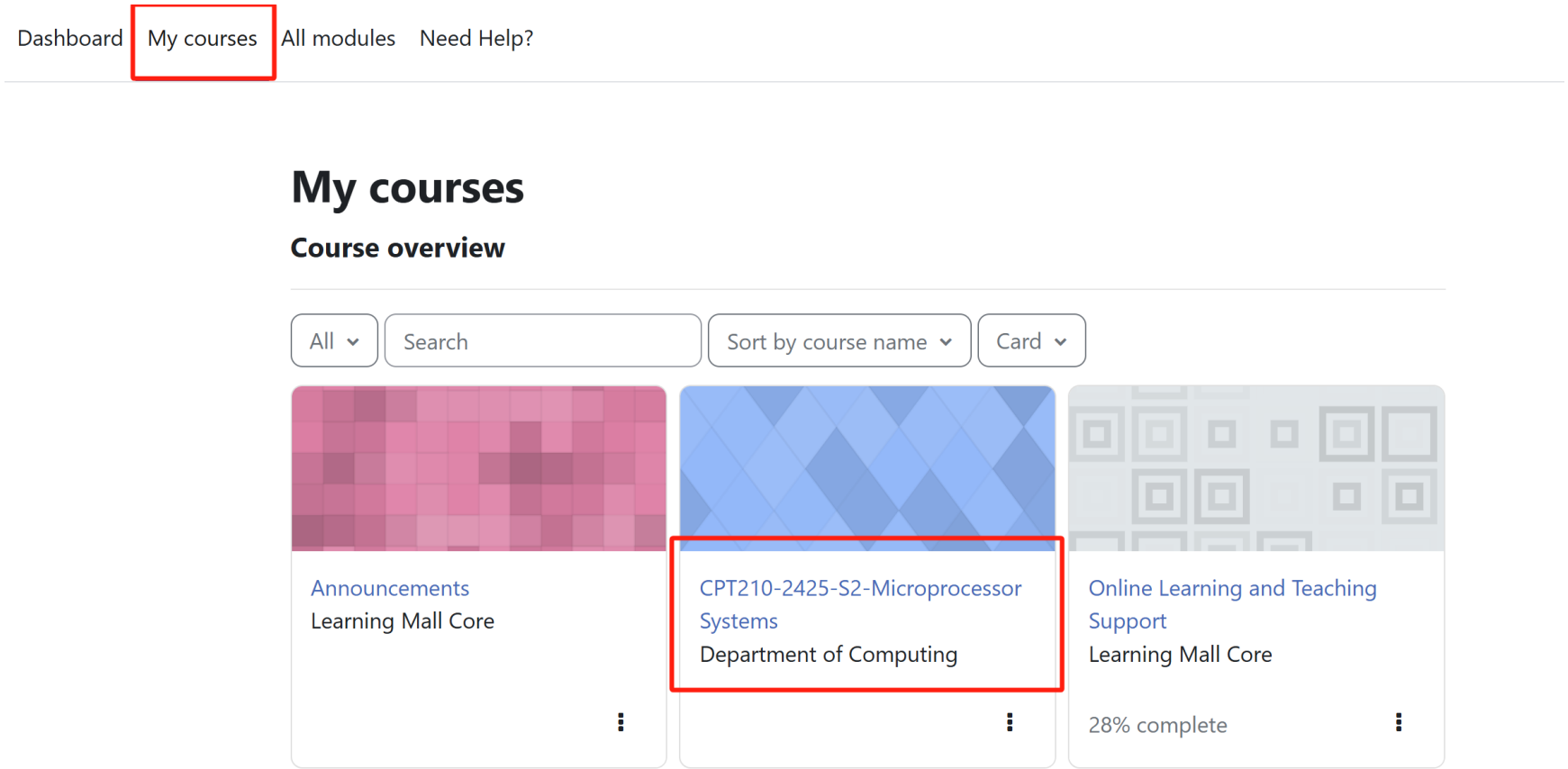
Instructions:

1. Open a browser, type the link and enter **LMO**: https://core.xjtlu.edu.cn/my/



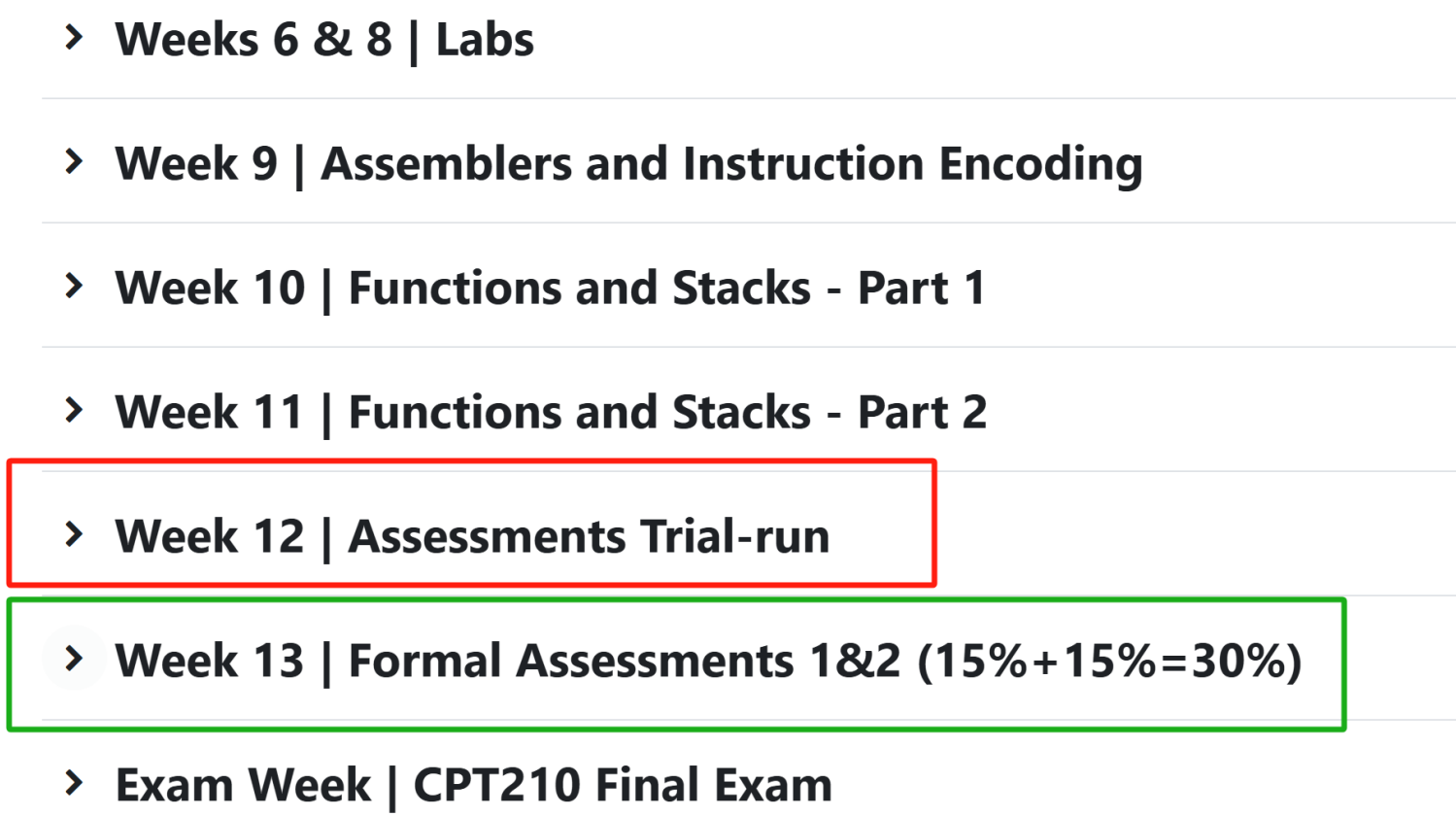
Instructions:

2. Click on **My courses** and enter **CPT210** by clicking



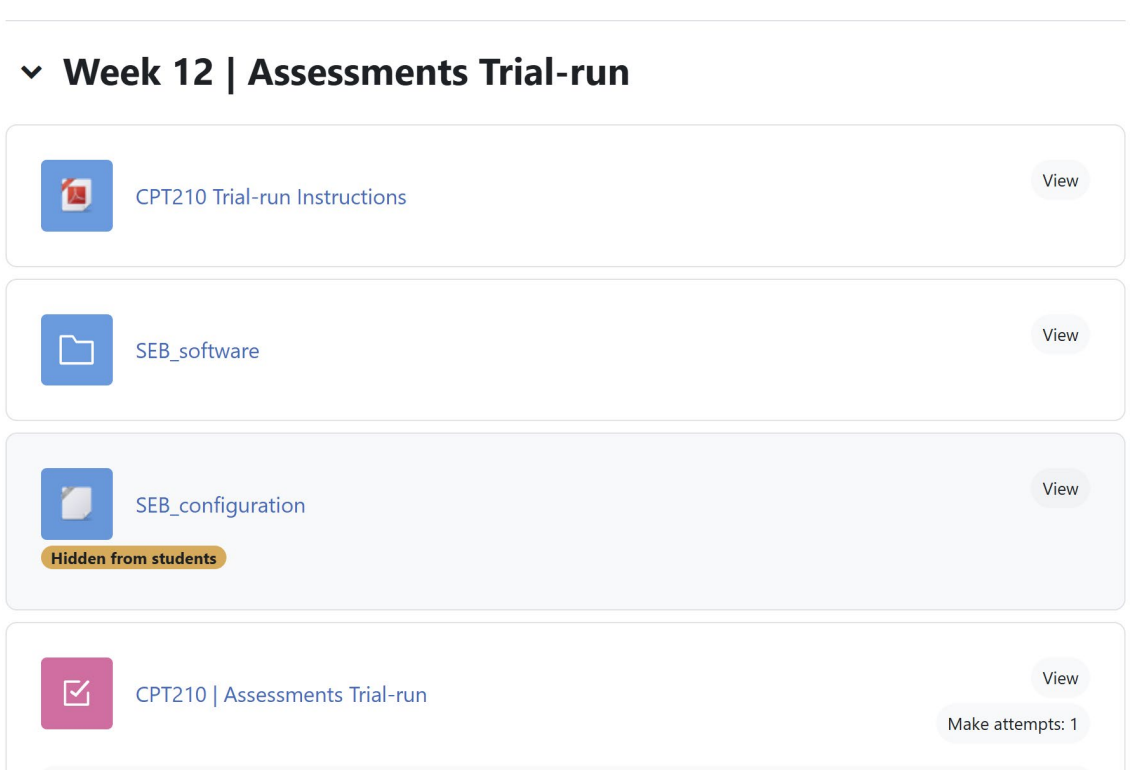
Instructions:

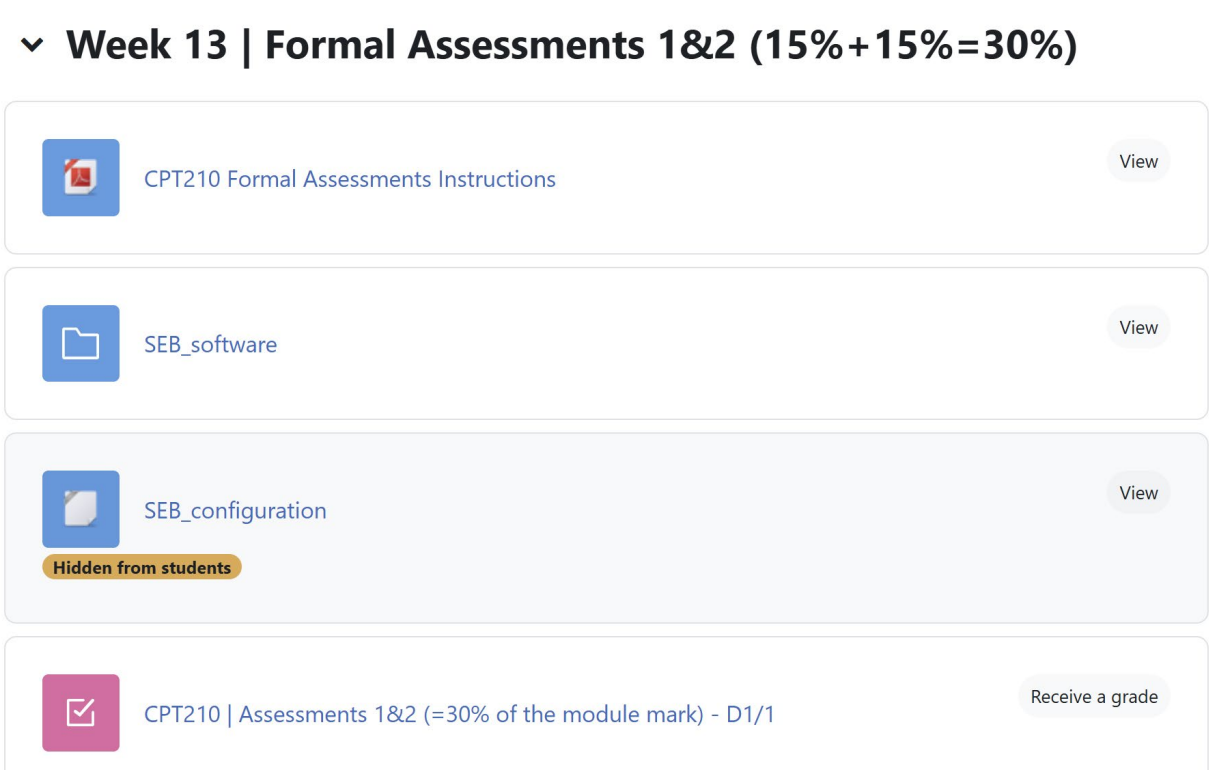
3. Find **Week 12 | Assessment Trial-run** / **Week 13 | Formal Assessment**



Instructions:

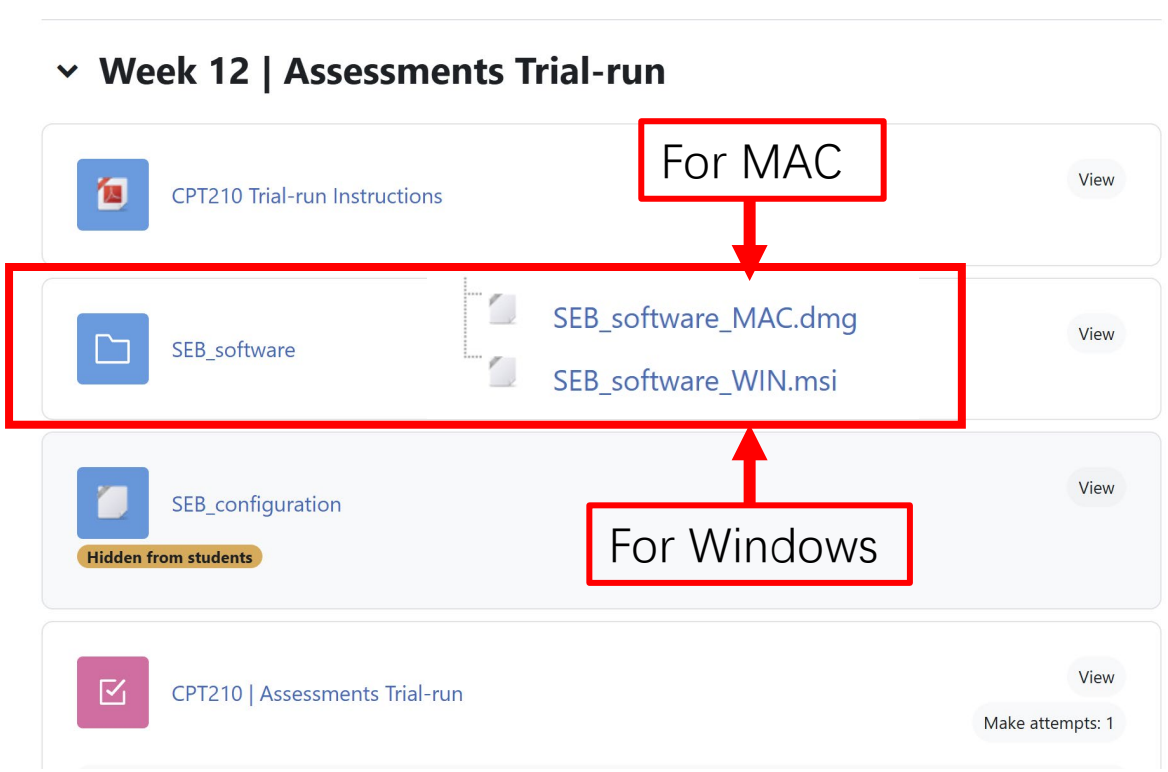
4. Click the arrow "**>**" in front to expand the tab

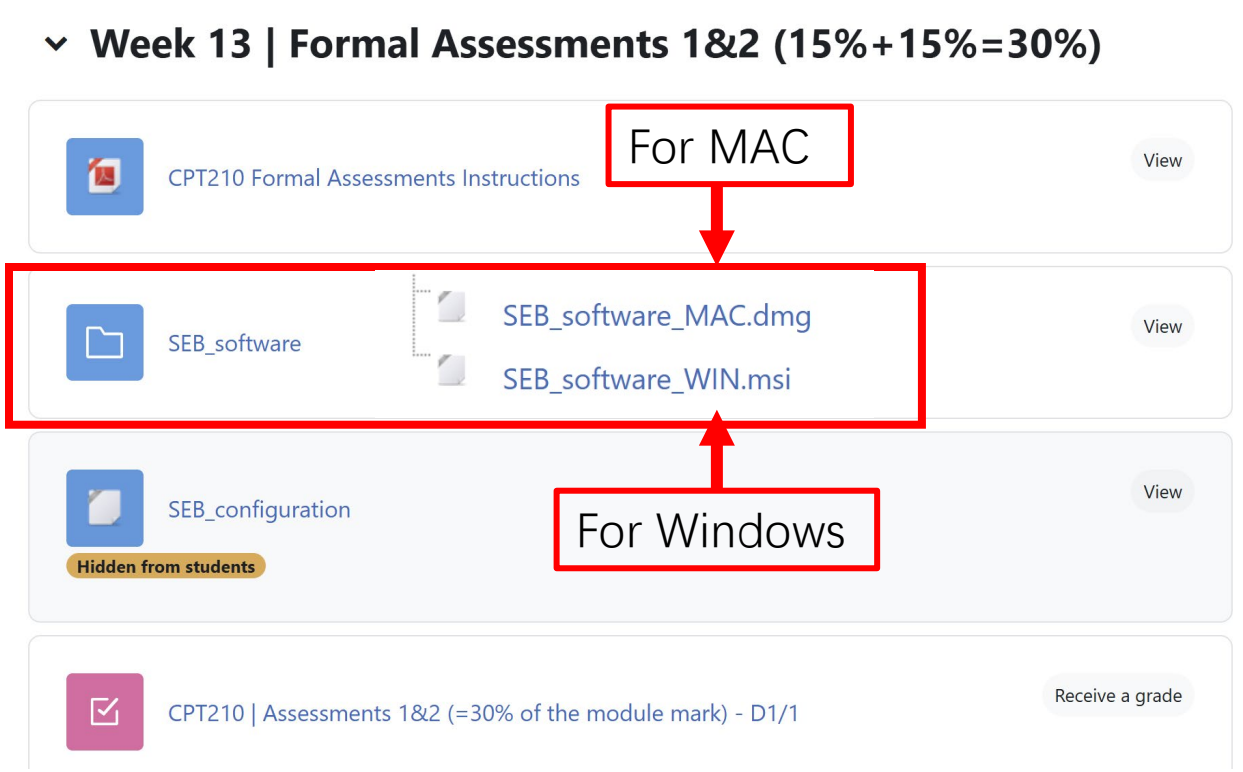




Instructions:

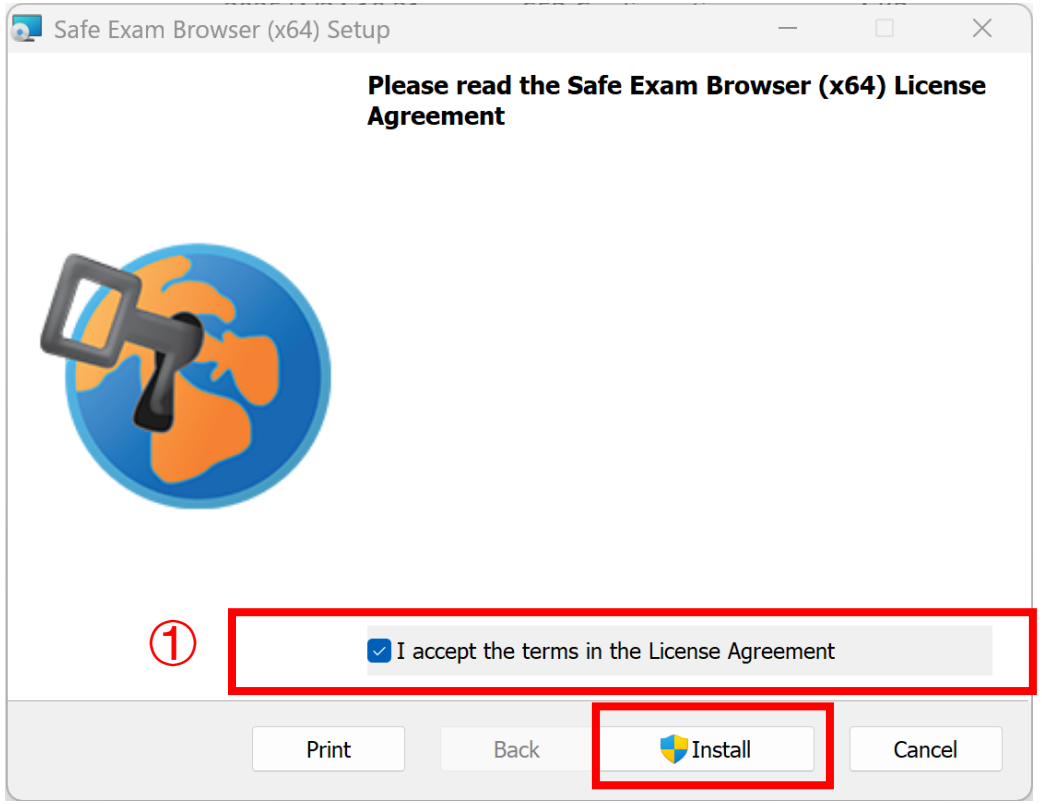
5. Download the **SEB_software** according to your OS by simple clicking it

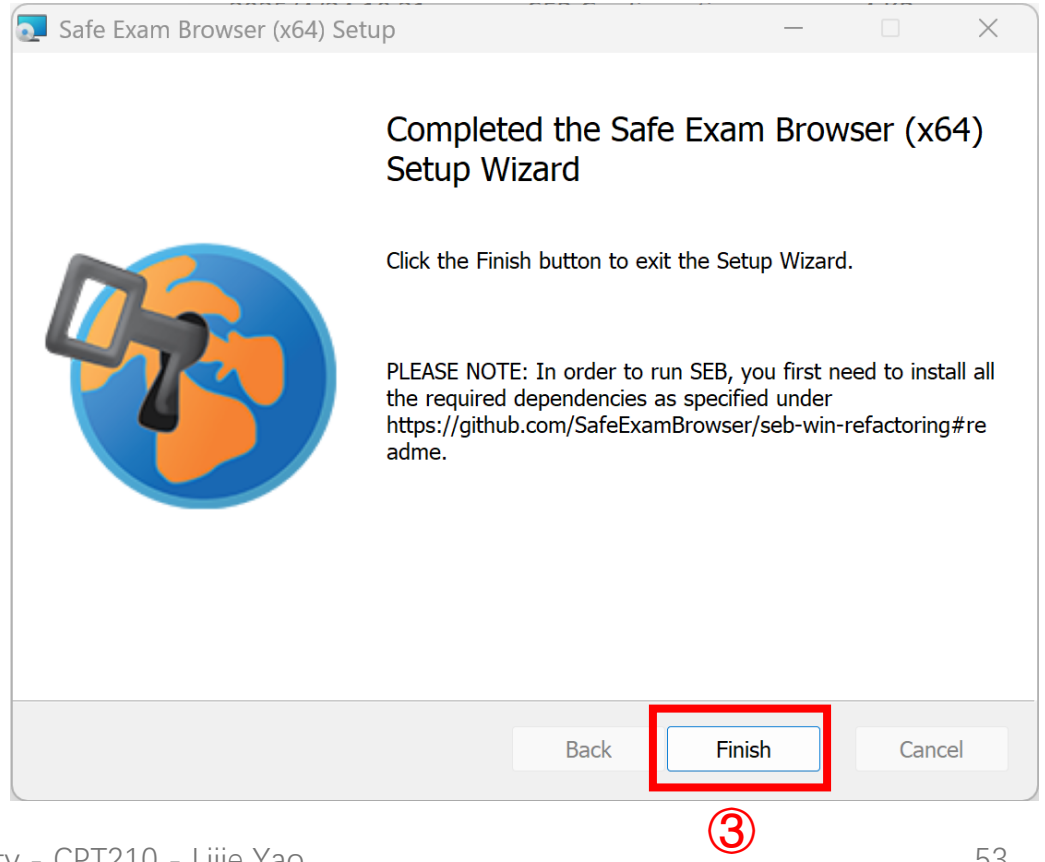




Instructions:

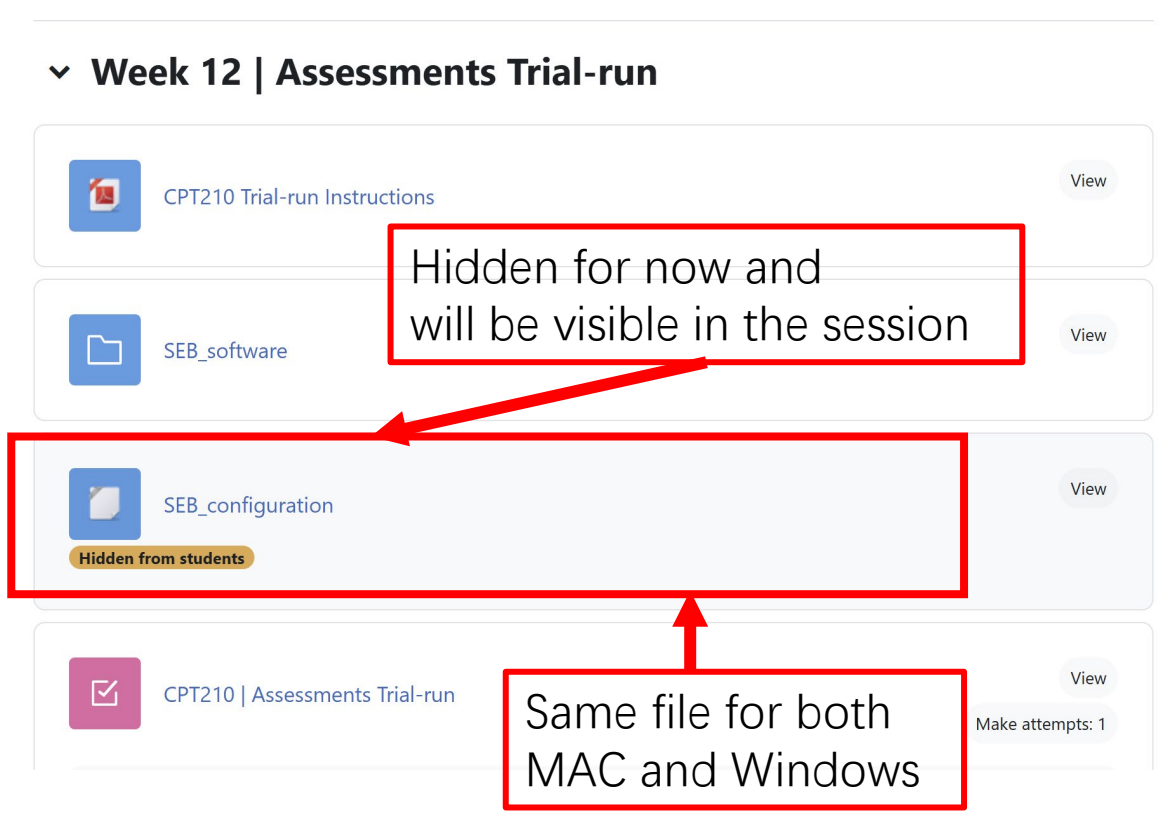
6. Install the **SEB_software** on your laptops by double-clicking, **accept** the terms, click on **Install**, and click on **Finish** once the installation completes.

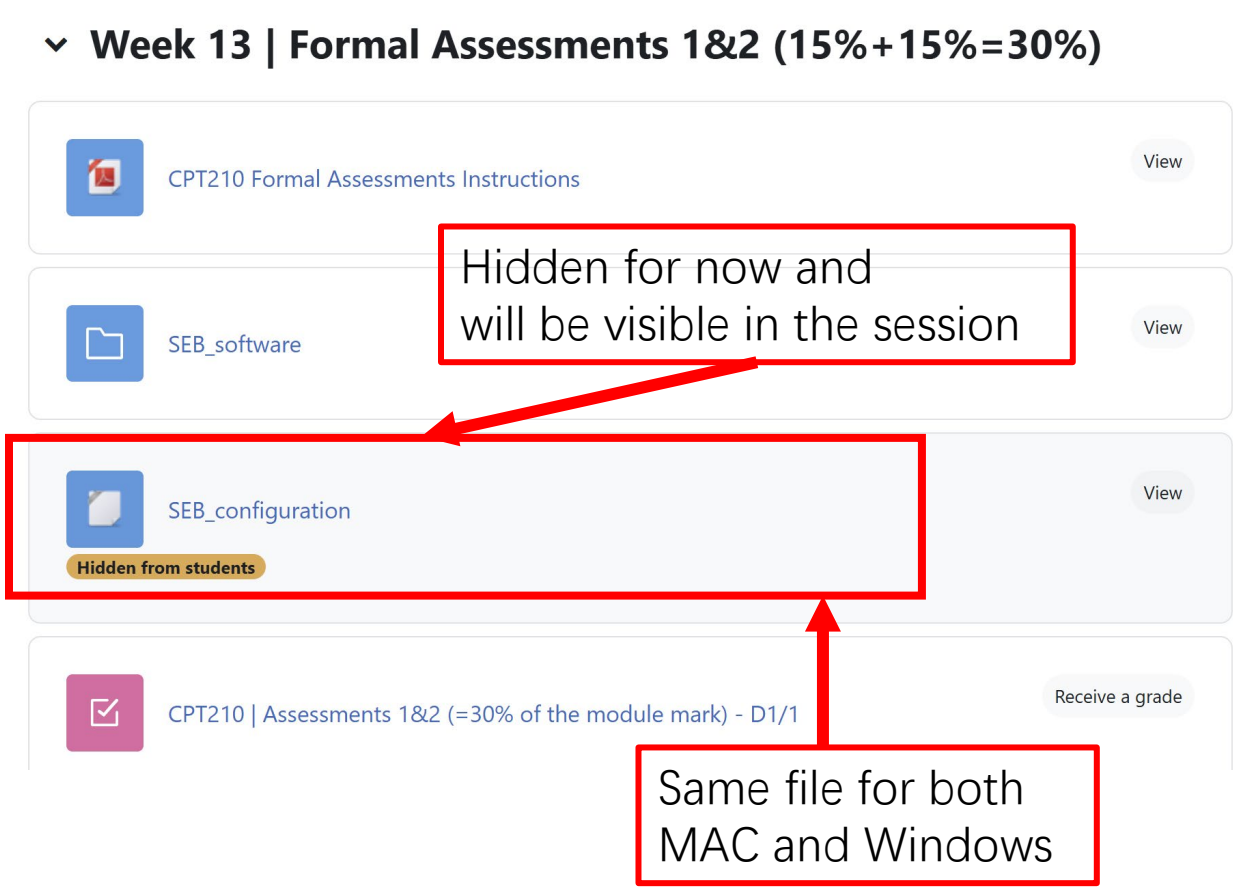




Instructions:

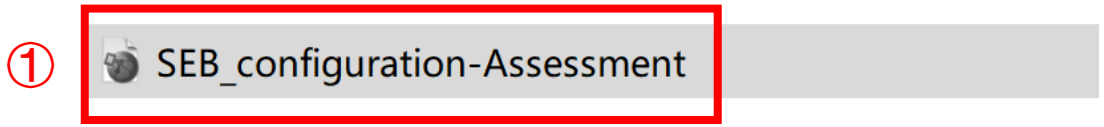
7. Go back to the **LMO**, download the **SEB_configuration** by simple clicking it

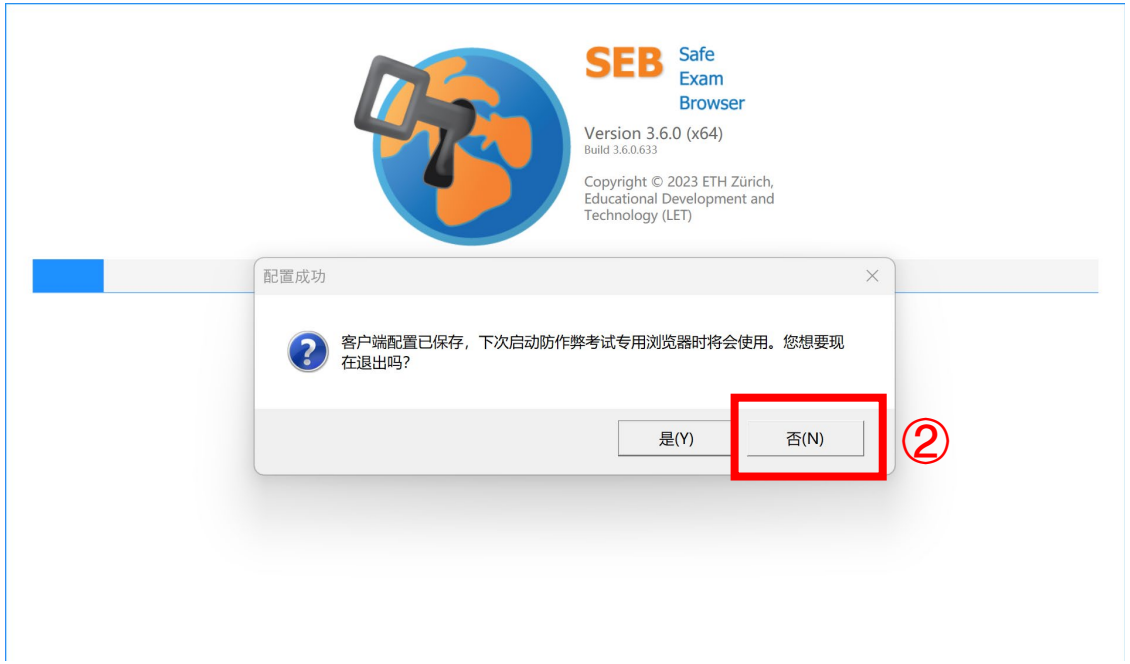




Instructions:

7. Double-click on the downloaded **SEB_configuration-Assessment**, make sure you have **internet connection**, click **No** for the pop-up question.

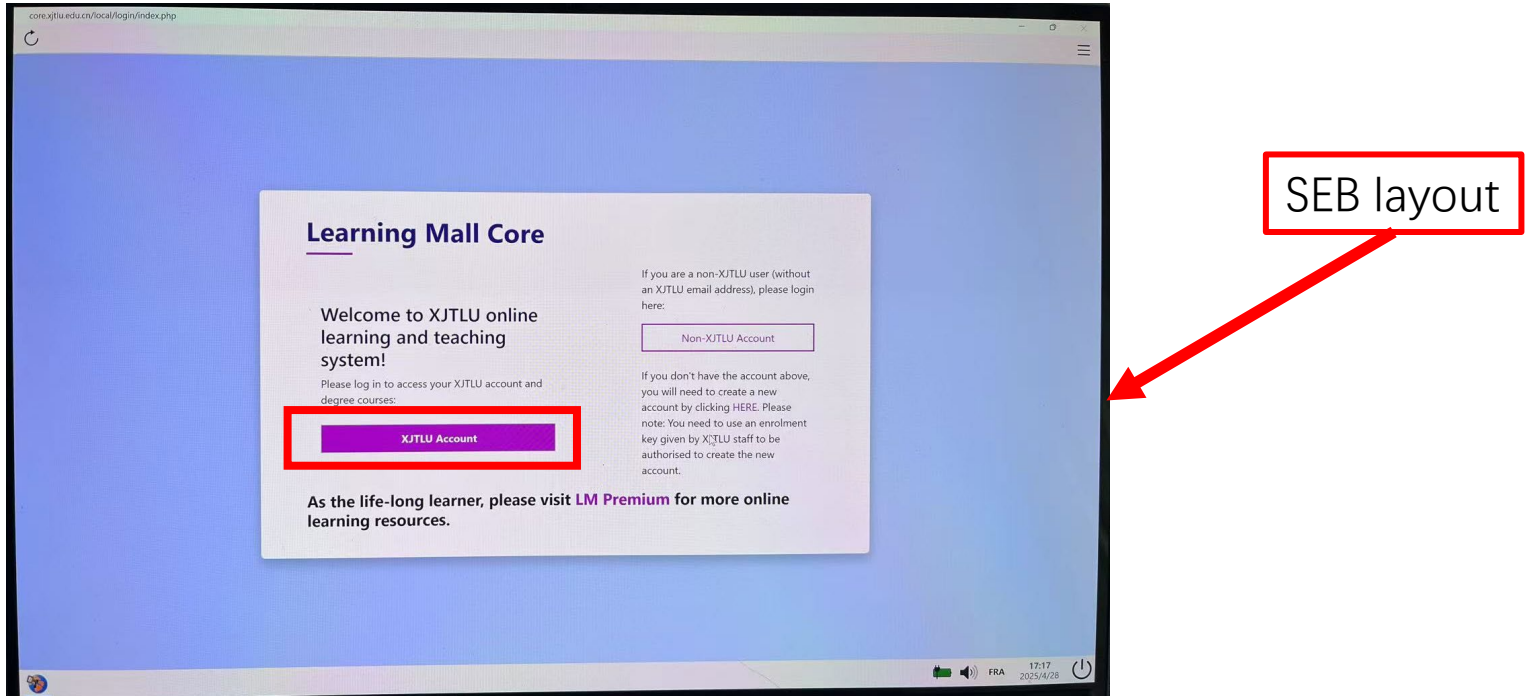




⚠️ *You will not be able to use SEB if you do not have a working internet connection, and/or if you select « YES» for the pop-up questions.*

Instructions:

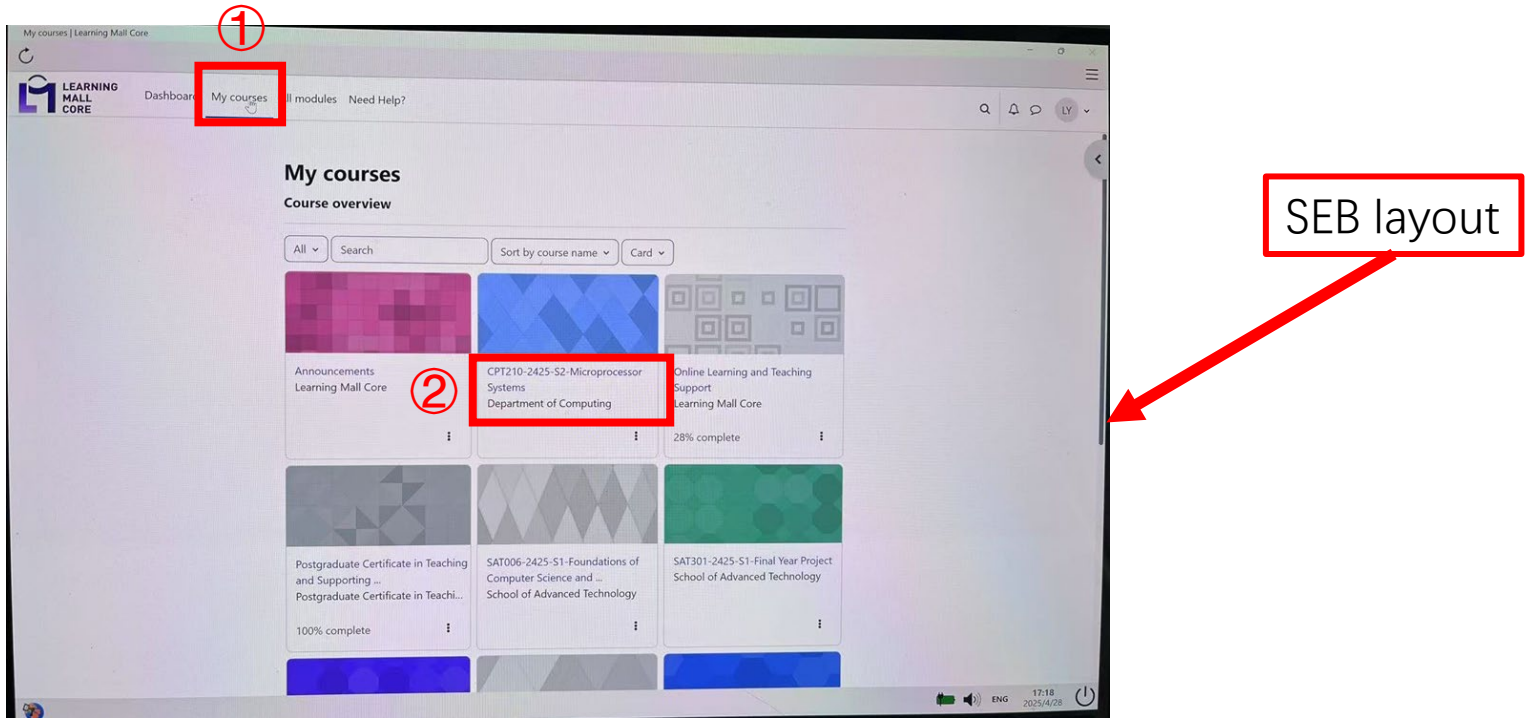
8. If you complete the previous steps correctly, you will enter the LMO in SEB automatically, click **XJTLU Account** to login.



⚠️ *You won't be able to access Trial-run/Assessments outside SEB. You will not be able to quit SEB without an exit code. In SEB, You will not be able to use clipboard and cannot leave LMO.*

Instructions:

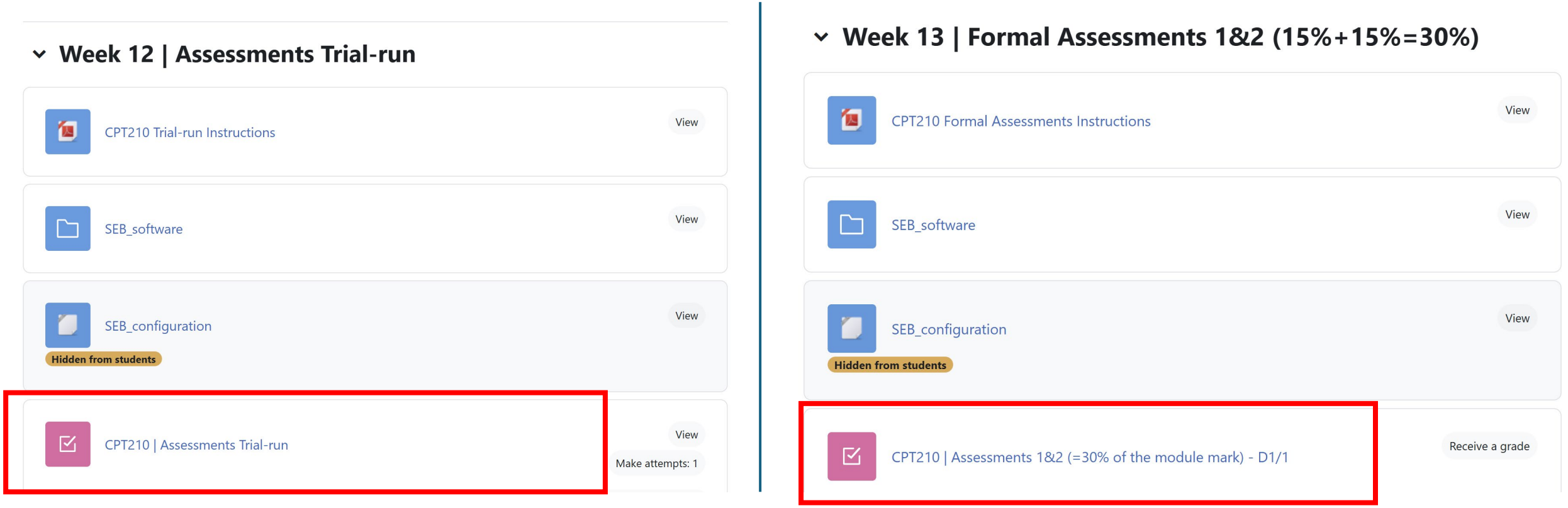
9. Click on **My courses** and enter **CPT210** again, by clicking**, in SEB**



⚠️ *You won't be able to access Trial-run/Assessments outside SEB. You will not be able to quit SEB without an exit code. In SEB, You will not be able to use clipboard and cannot leave LMO.*

Instructions:

10. Find again **Week 12 | Assessment Trial-run** / **Week 13 | Formal Assessment,** click on the **Assessments Trial-run** / **Assessment 1&2**

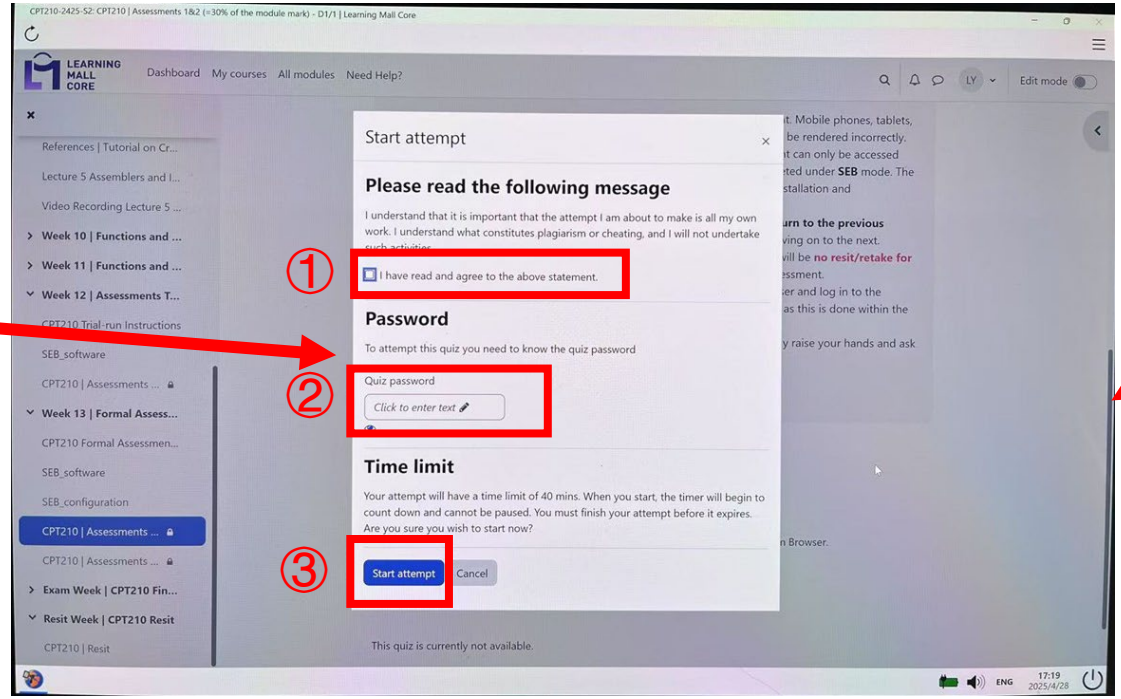


⚠️ *You won't be able to access Trial-run/Assessments outside SEB. You will not be able to quit SEB without an exit code. In SEB, You will not be able to use clipboard and cannot leave LMO.*

Instructions:

11. **Read** the statement, **enter the password**, click **Start attempt** to start your assessment. Once the assessment starts, the timer counts down.

Password will be given in class.

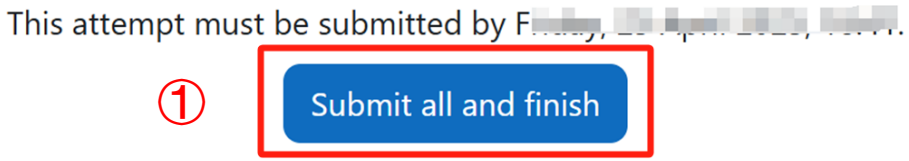


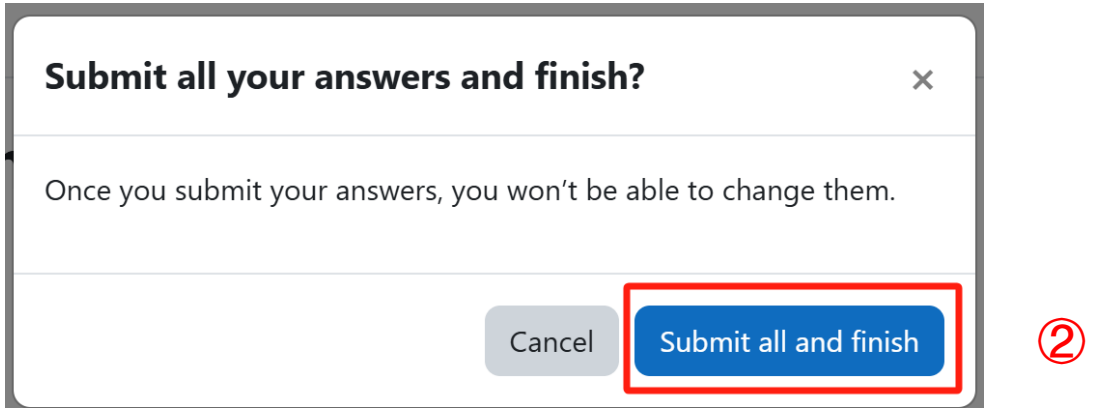
SEB layout

⚠️ *The timer only starts to count down when you start your attempt. However, the whole assessment is only available during a specific slot. You MUST start and complete your attempt within the slot. The attempt will be automatically submitted if the specific slot runs out.*

Instructions:

12. **Submit** your answers and **confirm** your submission once you finish.

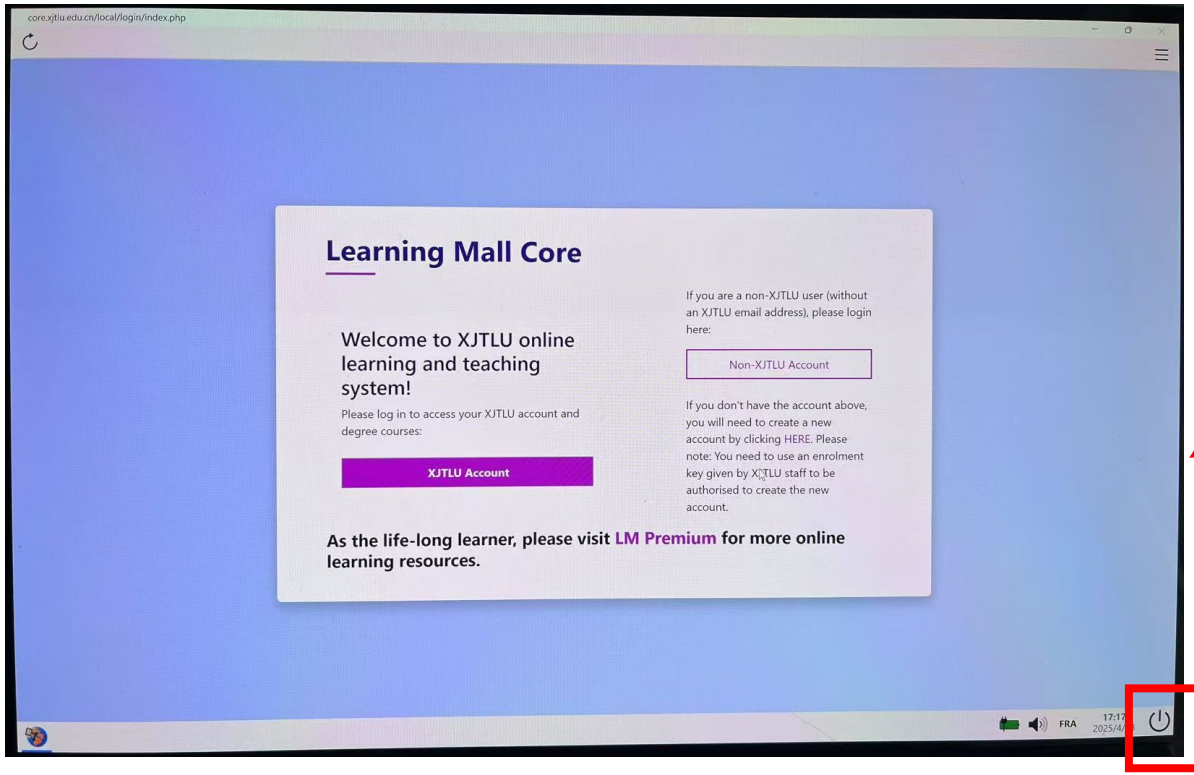




⚠️ *You may lose your answers/marks if you do not submit them.*

Instructions:

13. **Exit** the SEB with the given code.



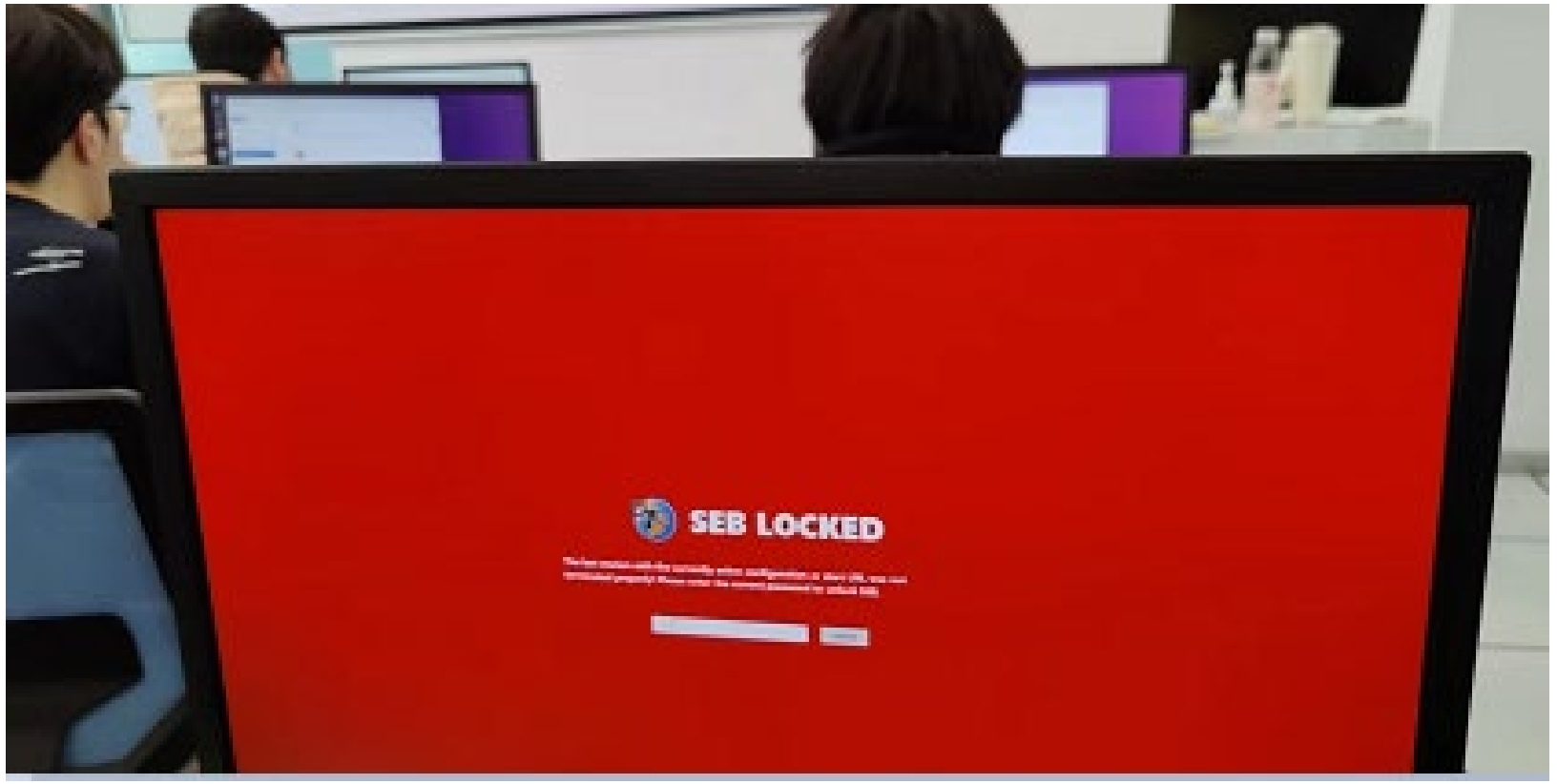
SEB layout

Exit button.
The exit code will be given at the end of the class.

⚠️ *You should wait for the exit code, otherwise you cannot exit SEB and your machine will not be able to work normally..*

Instructions:

If you try to leave or refer to other pages during the assessment, your laptop will be locked immediately:



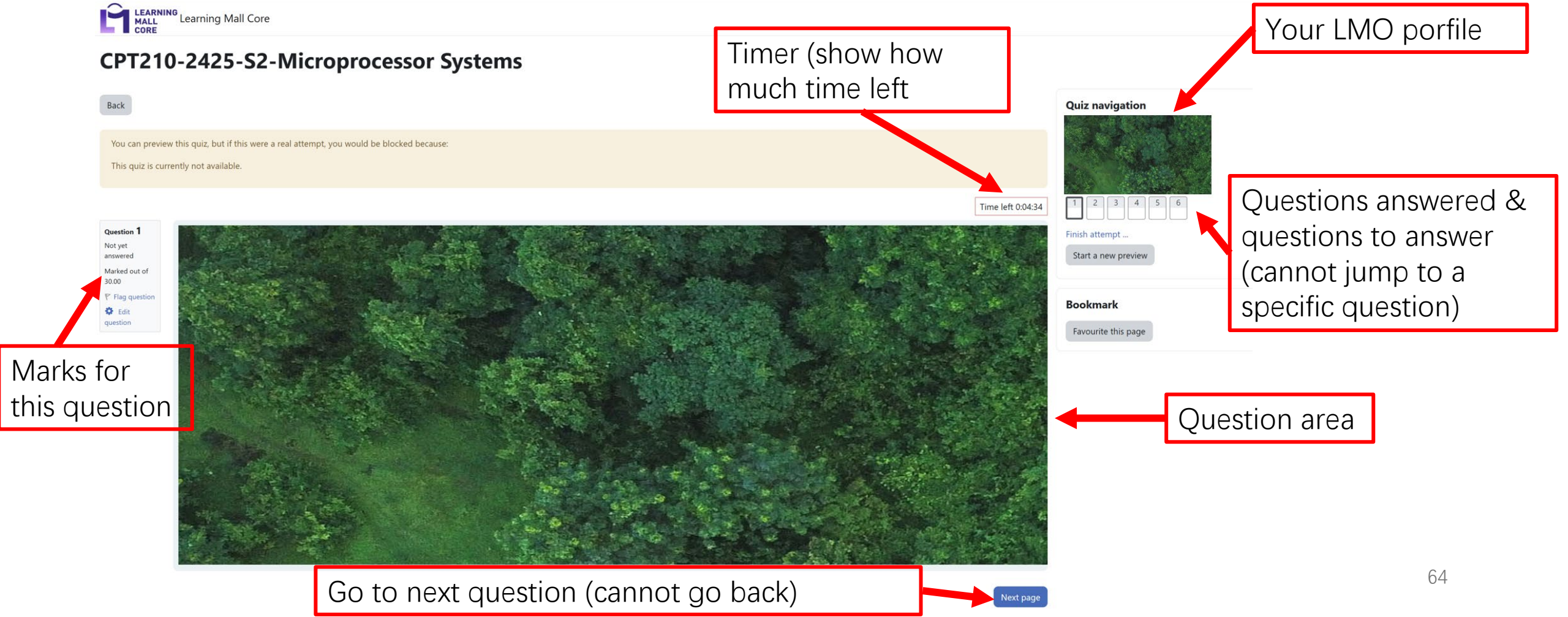
Structure:

1. How long time the assessment last: 40 minutes
2. When the assssment can be accessed: Open from half an hour after the starting of Tutorials
3. When the formal assessment is: May 19th
4. How to access the assessment: Under SEB, with correct password
5. Where the assessment will be: In-person (on-site) online
6. How many questions: 6 big questions
7. How many marks: 200 marks in total (for Assessment 1 and 2)
8. Percentage of the module mark: 30%
9. Mode: Close-book; only paper-based class notes are allowed
10. Range: Lectures 1-7 + Labs 1-2

NO RESIT OF ASSESSMENT

Structure:

How the question behaive: A single-way encoding, one page = one question, timer, marks questions in total are availale



More information is on LMO, CPT210 module page:

CPT210 | Assessments 1&2 (=30% of the module mark) - D1/1

Receive a grade

Opens: Friday, 16 May 2025, 09:30
Closes: Friday, 16 May 2025, 10:50

This **on-site online** Assessment only applies to students who belong to the **Tutorial D1/1 group**.

The Assessment is closed-book and will include 6 questions (200 marks in total, 1 question per page) extracted from Lectures 1-7 and Labs 1-2, lasting 40 minutes. It will open from 9:30 AM Beijing time on May 16th (Friday).

The Assessment can **ONLY be accessed within the determined slot, with the correct password (will be given in the Tutorial) and under SEB mode (follow the instructions BEFORE the assessment starts).**

The SEB mode cannot be exited once logged in.

IMPORTANT NOTES:

1. You should bring your own **fully-charged laptops** to take this assessment. Mobile phones, tablets, or any other mobile devices **CANNOT** support the questions as they may be rendered incorrectly.
2. You should follow the **CPT210 Assessments Instructions**. The assessment can only be accessed after successfully installing the **SEB** configuration and can only be completed under **SEB** mode. The assessment **CANNOT** be opened directly in Learning Mall, without SEB installation and configuration.
3. Each question is presented **sequentially**, which means **you CANNOT return to the previous questions**. Therefore, make correct answers for each question before moving on to the next.
4. If you **miss this assessment, you will lose 30% for this module**. There will be **no resit/retake for this assessment**. You will receive a mark of zero should you miss this assessment.
5. Should your laptops freeze while taking the assessment, restart the browser and log in to the assessment again. It should continue from where you last left off, as long as this is done within the duration of the assessment.
6. Finally, if you face any technical issues during the assessment, immediately raise your hands and ask TAs for help.

CPT210 | Assessments 1&2 (=30% of the module mark) - D1/2

Receive a grade

Opens: Friday, 16 May 2025, 11:30
Closes: Friday, 16 May 2025, 12:50

This **on-site online** Assessment only applies to students who belong to the **Tutorial D1/2 group**.

The Assessment is closed-book and will include 6 questions (200 marks in total, 1 question per page) extracted from Lectures 1-7 and Labs 1-2, lasting 40 minutes. It will open from 11:30 AM Beijing time on May 16th (Friday).

The Assessment can **ONLY be accessed within the determined slot, with the correct password (will be given in the Tutorial) and under SEB mode (follow the instructions BEFORE the assessment starts).**

The SEB mode cannot be exited once logged in.

IMPORTANT NOTES:

1. You should bring your own **fully-charged laptops** to take this assessment. Mobile phones, tablets, or any other mobile devices **CANNOT** support the questions as they may be rendered incorrectly.
2. You should follow the **CPT210 Assessment Instructions**. The assessment can only be accessed after successfully installing the **SEB** configuration and can only be completed under **SEB** mode. The assessment **CANNOT** be opened directly in Learning Mall, without SEB installation and configuration.
3. Each question is presented **sequentially**, which means **you CANNOT return to the previous questions**. Therefore, make correct answers for each question before moving on to the next.
4. If you **miss this assessment, you will lose 30% for this module**. There will be **no resit/retake for this assessment**. You will receive a mark of zero should you miss this assessment.
5. Should your laptops freeze while taking the assessment, restart the browser and log in to the assessment again. It should continue from where you last left off, as long as this is done within the duration of the assessment.
6. Finally, if you face any technical issues during the assessment, immediately raise your hands and ask TAs for help.

Formal assessment on Week 13

Assessment trial-run on Week 12

We are here

Week 2	Week 3	Week 4	Week 5	Week 6	Week 8	Week 9	Week 10	Week 11	Week 12	Week 13
Lecture 1	Lecture 2	Lecture 3	Lecture 4	Lab 1	Lab 2	Lecture 5	Lecture 6	Lecture 7	Lecture 8	Tutorial 2 Formal Assessment
									Tutorial 1 Assessment Trial-run	

Assessment 1&2 will be combined into one and should be completed at **a specific time in the Tutorial**

Assessment 1 (CW 1) 15%	Assessment 2 (CW 2) 15%
Final Exam 70%	

Come in person and don't miss!!