Computer Architecture I Mid-Term I

Chinese Name: ______________________________

Pinyin Name: ______________________________

Student ID: ______________________________

E-Mail ... @shanghaitech.edu.cn: ______________________________

Question:	1	2	3	4	5	6	7	8	9	10	11	12	Total
Points:	1	3	11	11	10	12	7	16	5	10	7	7	100
Score:													

- This test contains 21 numbered pages, including the cover page, printed on both sides of the sheet.
- We will use Gradescope for grading, so only answers filled in at the obvious places will be used.
- Use the provided blank paper for calculations and then copy your answer here.
- Please turn **off** all cell phones, smartwatches, and other mobile devices. Remove all hats and headphones. Put everything in your backpack. Place your backpacks, laptops and jackets out of reach.
- Unless told otherwise always assume a 32bit machine.
- The total estimated time is 120 minutes.
- You have 120 minutes to complete this exam. The exam is closed book; no computers, phones, or calculators are allowed. You may use two A4 pages (front and back) of handwritten notes in addition to the provided green sheet.
- There may be partial credit for incomplete answers; write as much of the solution as you can. We will deduct points if your solution is far more complicated than necessary. When we provide a blank, please fit your answer within the space provided.
- Do **NOT** start reading the questions/ open the exam until we tell you so!

1 1. **First Task (worth one point): Fill in you name**

Fill in your name and email on the front page and your ShanghaiTech email on top of every page (without @shanghaitech.edu.cn) (so write your email in total 21 times).

2. **Various Questions**

3 (a) Name the 6 Great Ideas in Computer Architecture as taught in the lectures.

Solution:

1. Abstraction (Layers of Representation/Interpretation)
2. Moore's Law (Designing through trends)
3. Principle of Locality (Memory Hierarchy)
4. Parallelism
5. Performance Measurement and Improvement
6. Dependability via Redundancy

3. **Number Representation**

3 (a) Given the number `0x811F00FA`. It can be interpreted as:

a binary number: __________

four unsigned bytes: __________

four two's complement bytes: __________

Solution:
1000 0001 0001 1111 0000 0000 1111 1010;
129, 31, 0, 250; −127, 31, 0, −6;

4 (b) A **quarter** is a single byte split into the following fields (1 sign, 3 exponent, 4 mantissa): `SEEEMMMM`. It has all the properties of **IEEE 754** (including denormal numbers, NaNs and $\pm\infty$) just with different ranges, precision and representations.

For a **quarter**, the bias of the exponent is 3, and the implicit exponent for denormal numbers are -2.

What is the largest number smaller than ∞?

In binary ______________________

In decimal ______________________

Which negative denormal number is closest to 0?

In binary ______________________

In decimal ______________________

Solution: 01101111; 15.5; 10000001; $-\frac{1}{64}$.

4 (c) What is the value of `q1`, `q2`, `c`, `d`?

Hint Rounding mode: round toward even/0.

```
quarter q1, q2, q3, c, d;
q1 = -0.25;
q2 = -4.0;
q3 = 0.125;
c = q1 + (q2 + q3);
d = (q1 + q2) + q3;
```

`q1` in binary ______________________

`q2` in binary ______________________

`c` in decimal ______________________

`d` in decimal ______________________

Solution: 10010000; 11010000; −4.25; −4.50.

4. **C Basics**

5 (a) Memory of C

```c
#include <stdlib.h>

int main() {
    static int p = 5;

    char *str = ______________________;
    /* some other codes, and you can skip it. */
    return 0;
}
```

1. You need to allocate a string `str` containing `p` characters. Write the code above (please use `malloc`).

> **Solution:** `char *str = malloc(sizeof(char) * (p + 1));`

2. Fill in the correct memory section based on what the given C expressions evaluate to.

`&p` ________________________

`&str` ________________________

`str` ________________________

> **Solution:** static, stack, heap.

3 (b) Catch bugs!

1. When you want to debug with GDB, what flag you will put in your compilation?

   ____

   Solution: -g

2. Write down some essential commands in GDB. Example: Start your program: `run`/`r`

   Set break point: ______________________________

   Show next line(stepping into function calls): ______________________________

   Solution: `break`/`b`, `step`/`s`

3 (c) C programming: Reverse singly linked list. For example, convert $1 \to 2 \to 3 \to$ `NULL` to $3 \to 2 \to 1 \to$ `NULL`. (You may not need all of the lines)

```c
#include <stdio.h>
#include <stdlib.h>

/* Definition for singly-linked list. */
struct ListNode {
    int val;
    struct ListNode *next;
};

/* Given the head of a singly linked list, reverse
    the list, and return the head of reversed list.*/
struct ListNode *reverse_list(struct ListNode *head) {
    struct ListNode *prev = NULL;
    struct ListNode *curr = head;
    struct ListNode *next = head;
    while (curr) {
        next = next->next;

        ____________________________

        ____________________________

        ____________________________

        ____________________________

        ____________________________

        ____________________________
    }
    return prev;
}
```

Solution:

```
curr->next = prev;
prev = curr;
curr = next;
```

5. **Byte-Swap Operation**

Assuming we are in a **32bit**, **little endian** system. *Little Dragon* receives a 4-byte integer num, he wants to swap the value of num's i^{th} byte and j^{th} byte ($i, j \in \{0, 1, 2, 3\}$, $i \neq j$) to get a new number!

$\boxed{3}$ (a) **Idea I**: *Little Dragon* wants to directly retrieve the i^{th} and j^{th} byte of num, then swap them.

First of all, define a MACRO to get the i^{th} byte of num. Read the following C code, then help *Little Dragon* to fill in the blank lines (Line 4 and 10) so the output should be `0x34`. When defining the MACRO, use `&`, `|`, `^`, `~`, `>>`, `<<` operators **only**. Remember to write a meaningful MACRO such that *Little Dragon* can reuse it again (directly return `0x34` is not allowed)!

```
1 #include <stdio.h>
2 #include <stdint.h>
3
4 #define GET_BYTE(num, ind) ________________________________
5
6 int main(){
7     int number, index;
8     int8_t byte;
9     number = 0x12345678;
10
11     index = ___________; /* index is one of {0, 1, 2, 3} */
12     byte = GET_BYTE(number, index);
13     printf("%#x\n", byte); /* should print 0x34 */
14     return 0;
15 }
```

Write your answer above.

Solution: `(((num) >> ((ind) << 3)) & 0xFF)`; 2.

Notice that there should be brackets around `num` and `ind`!

4 (b) **Idea II**: An alternative way to fetch the i^{th} byte is *Union*. *Little Dragon* wrote the following code, but he is a little confused about the concept of *little endian* and *big endian*. Help him answer the questions below!

```
1 #include <stdio.h>
2 #include <stdint.h>
3
4 /* Tip on union: data type that stores its members
5    in the same memory location */
6 typedef union {
7     struct {
8         uint8_t byte0;
9         uint8_t byte1;
10        uint8_t byte2;
11        uint8_t byte3;
12    } bytes;
13    int all_bits;
14 } MyInt;
15
16 int main() {
17    MyInt intA;
18    intA.all_bits = 0x12345678;
19    printf("%#x, %#x\n", intA.bytes.byte1, intA.bytes.byte3);
20    return 0;
21 }
```

What is the expected output (in hexadecimal format) of Line 19:

- if the system is **little endian**?

____

- if the system is **big endian**?

____

Solution:

0x56, 0x12;

0x34, 0x78.

3 (c) **Idea III**: *Little Dragon* is fasczinated in playing with bitwise operations. He wrote the following function in C.

```
void byte_xor(int num, int a, int b) {
    char *ret_val = (char *) &num;

    ret_val[b] ?? ret_val[a];
    ret_val[a] ?? ret_val[b];
    ret_val[b] ?? ret_val[a];

    printf("%#x\n", num);
}
```

What operators are expected to substitute the `??` in Line 4, 5, and 6, such that the result of `byte_xor(0x1133CCFF, 1, 3)` will be `0xCC3311FF`?

A. `&=, &=, &=`   B. `&=, ^=, ^=`   C. `|=, ^=, ~=`   D. `^=, ^=, ^=`

Solution: D

6. **RISC-V programming**

In this question, you are asked to implement a simple recursive function in RISC-V. The function takes a decimal number as input, then outputs it's octal representation encoded as decimal digits. For example, if the input to this function is 100, then the output would be 144.

The recursive function implemented in C is given below:

```c
int find_octal(unsigned int decimal) {
    if (decimal == 0) {
        return 0;
    } else {
        return decimal % 8 + 10 * find_octal(decimal / 8);
    }
}
```

A skeleton of RISC-V code is given below.

DO NOT fill in them immediately. Do some warm-ups first!

```
find_octal:
    addi    sp, sp, -8
    sw      ra, 4(sp)
    sw      s0, 0(sp)

    beq     a0, x0, _____

    _____________________  # set s0 to something

    _____________________  # set a0 to something

    jal     ra, _________  # recursive call

    _____________________
    mul     a0, t0, a0

    _____________________  # a0 = ???
postamble:

    _____________________  # Restore ra

    _____________________  # restore ...

    _____________________  # restore ...
end:
    jr      ra
```

$\boxed{2}$ (a) Translate the following RISC-V instructions into machine code.

```
sw      ra, 4(sp)    ______________________

andi    s0, a0, 7    ______________________
```

Solution:

```
0x00112223
0x00757413
```

[2] (b) What is one pseudo instruction in the RISC-V code above? How can you change it into one base instruction?

Pseudo instruction: ____________________

After your change: ____________________

Solution:

```
jr      ra
jalr    x0 ra
```

[8] (c) Fill in the missing code above.

Solution:

```
beq     a0, x0, postamble
andi    s0, a0, 7
srli    a0, a0, 3
jal     ra, find_octal
li      t0, 10 / addi    t0, x0, 10
add     a0, a0, s0
lw      ra, 4(sp)
lw      s0, 0(sp)
addi    sp, sp, 8
```

7. RISC-V Basic

[5] (a) Write a function in RISC-V code to return 0 if the input 32-bit float is an infinite value, else a non-zero value. The input and output will be stored in `a0`, as usual. Do not use pseduo instructions!

```
is_not_infinity:
```

```
ret  #  <=  Return instruction
```

Solution:

```
is_not_infinity:
        lui a1 0x80000
        addi a1 a1 -1
        and a0 a0 a1
        lui a1 0x7F800
        xor a0 a0 a1
    ret
```

2 (b) True or False.

1. Let `a0` point to the start of an array `x`. `lw s0, 4(a0)` will always load `x[1]` into `s0`.
2. After calling a function and having that function return, the t registers may have been changed during the execution of the function, while a registers cannot.

1	2

Solution:

1. False. This only holds for data types that are four bytes wide, like int or float. For data-types like char that are only one byte wide, `4(a0)` is too large of an offset to return the element at index 1, and will instead return a char further down the array (or some other data beyond the array, depending on the array length).

2. False. `a0` and `a1` registers are often used to store the return value from a function, so the function can set their values to the its return values before returning. So all a registers can be used temporary registers, as specified in the green card.

8. **CALL**

Answer the following questions with regard to the following C program.

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    if (argc == (1 + 1)) {
        printf("Hello, %s.\n", argv[1]);
    } else {
        printf("Goodbye.\n");
    }

    return 0;
}
```

$\boxed{8}$ (a) Select which stage of CALL is responsible for the following actions. Please fill you answer (A, B, C or D) in the table below.

A. Compiler B. Assembler C. Linker D. Loader

1. Removes all pseudo instructions.
2. Provide the address to the string `"Goodbye.\n"`.
3. Remove most duplicate instructions in the program in order to optimize the program.
4. Put arguments in the address of `argv` so that the program could read from it.
5. Incorporating dynamic libraries so that the program could call `printf` in the C standard library.
6. Creates the symbol table so that we can know the address to the function `main` in future stages.
7. The parser is used to determine the operator precedence in `argc == (1 + 1)`.
8. Determine the jump address the `if` statement is jumping to.

1	2	3	4	5	6	7	8

Solution: B; C; A; D; D; B; A; B.

$\boxed{8}$ (b) True or False. Please fill your answer (T or F) in the table below.

1. Pseudo instructions are not allowed in the output of compiler.
2. Statically-linked libraries are incorporated into the program during the load stage.
3. Dynamically-linked libraries are incorporated into the program during the link stage.
4. The interpreted program (like Python) runs way faster than a compiled one (like C) in most cases.

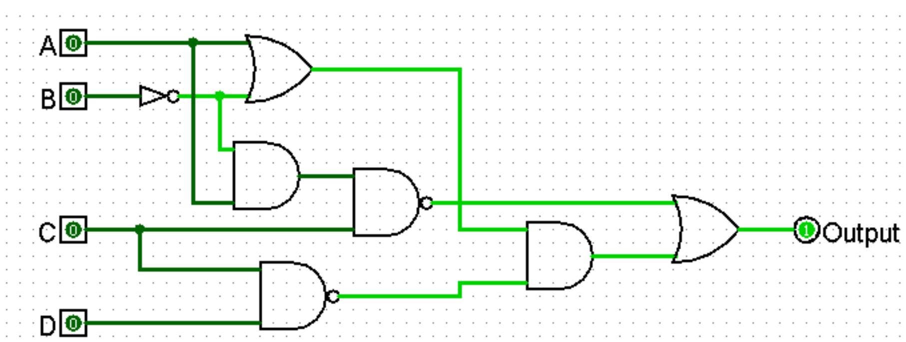
5. The assembler takes two passes over the code to resolve PC-Relative target addresses.
6. Copying arguments passed to the program onto the stack is done during the linking stage.
7. Assembler can always provide the correct immediate value when translating all `la` instructions.
8. Compiling stage is the one most often responsible for code optimization.

1	2	3	4	5	6	7	8

Solution: F; F; F; F; T; F; F; T.

9. **Logic**

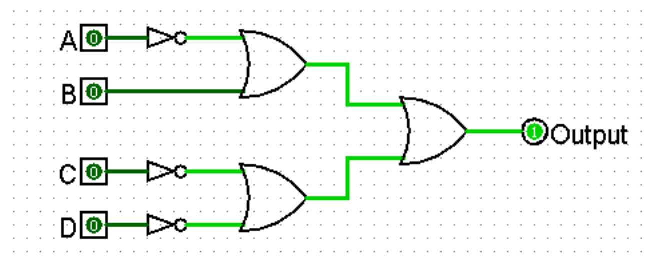
5 (a) The circuit shown below can be simplified. Please **write** down the boolean expression that exactly corresponds to the circuit shown (no simplification). Then simplify this prepossession step by step, applying one rule at a time. Then **draw** the circuit according to the simplified boolean expression using the minimum number of **one-** or **two-input** logic gates.



Solution:

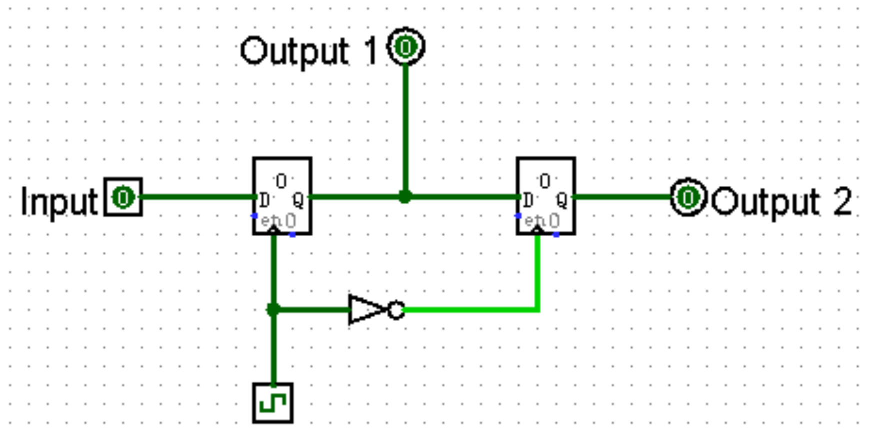
$$\begin{aligned}\overline{A\overline{B}C} + (A+\overline{B})(\overline{CD}) &= \overline{A} + B + \overline{C} + (A+\overline{B})(\overline{C}+\overline{D}) \\ &= \overline{A} + B + \overline{C} + A\overline{C} + A\overline{D} + \overline{B}\,\overline{C} + \overline{B}\,\overline{D} \\ &= \overline{A} + B + \overline{C} + A\overline{D} + \overline{B}\,\overline{D} \\ &= \overline{A} + B + \overline{C} + \overline{D}\end{aligned}$$

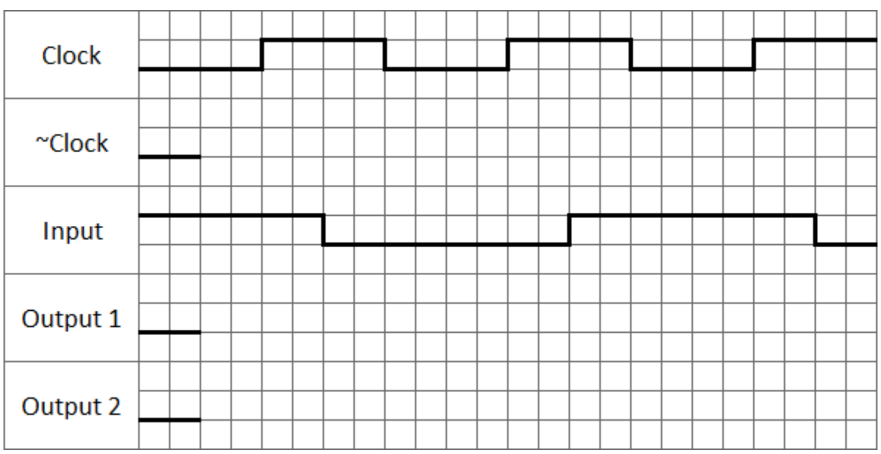
One possible circuit:

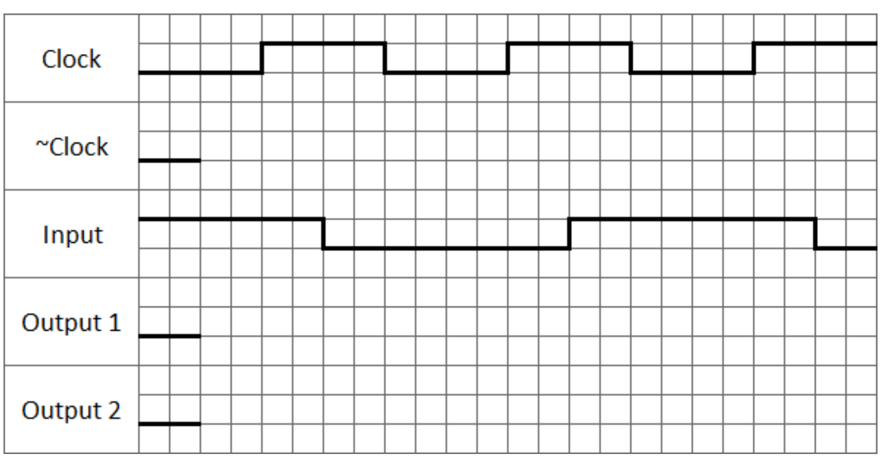

A
B
C
D
Output

10. **SDS**

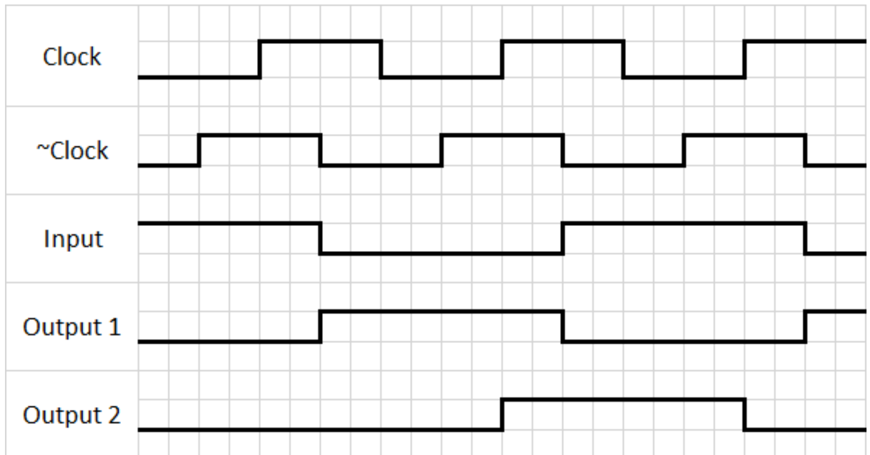
6 (a) Draw the Timing Diagram for the circuit below. NOT gates have a 2 ns propagation delay. For each register, the clk-to-q delay is 2 ns and setup time is 2 ns. The clock period is 8 ns, and each grid in the following diagram is a unit of 1 ns. The initial values of clock and output are given in the diagram. Use any of the two empty graphs to put in your answer (so you can re-do it). Clearly **mark your final answer** if you use more than one graph!



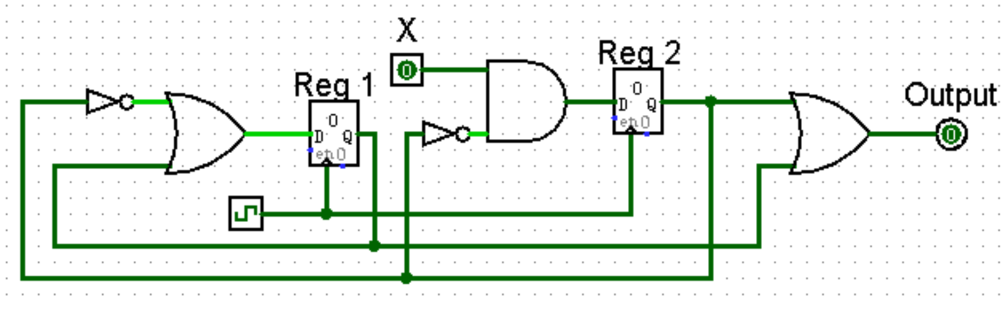




Solution:



$\boxed{4}$ (b) Consider the following circuit. Assume the clock has a frequency of 50 MHz, all gates have a propagation delay of 6 ns, X changes 10 ns after the rising edge of clk, Reg1 and Reg2 have a clk-to-q delay of 1 ns.



What is the **longest possible setup time** such that there are no setup time violations?

What is the **longest possible hold time** such that there are no hold time violations?

Solution:

The clock period is $\frac{1}{50\times10^6}s = 20\ ns$.

Reg1 longest possible setup time: the path is *the output of Reg*2 $\rightarrow$ *NOT* $\rightarrow$ *OR*, with a delay of 1 ns + 6 ns + 6 ns =13 ns. So 20 - 13 = 7 ns.

Reg2 longest possible setup time: the path is *X changes* $\rightarrow$ *AND*, with a delay of 10 ns + 6 ns = 16 ns. So 20 - 16 = 4 ns.

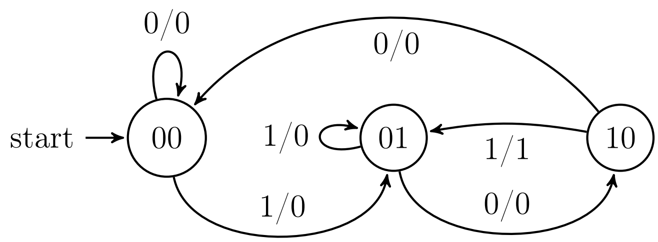
So longest setup time: min(7 ns, 4 ns) = 4 ns.

Reg1 longest possible hold time: the path is *the output of Reg*1 $\rightarrow$ *OR*, with a delay of 1 ns + 6 ns = 7 ns.

Reg2 longest possible hold time: the path is *the output of Reg*2 $\rightarrow$ *NOT* $\rightarrow$ *AND*, with a delay of 1 ns + 6 ns + 6ns = 13 ns.

So longest hold time: min(7 ns, 13 ns) = 7 ns.

11. **FSM**



3 (a) Fill in the truth table for the FSM above.

state bit1	state bit0	input	next state bit1	next state bit0	output
0	0	0			
0	0	1			
0	1	0			
0	1	1			
1	0	0			
1	0	1			

Solution:

state bit1	state bit0	Input	next state bit1	next state bit0	Output
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	1	0	0
0	1	1	0	1	0
1	0	0	0	0	0
1	0	1	0	1	1

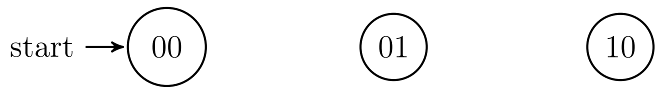
1 (b) What does the given FSM output with the input bit string '0100101010'?

Solution: It outputs '0000001010'.

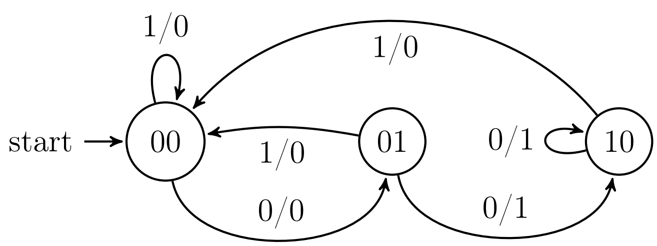
1 (c) What does the given FSM implement (Describe when the FSM will output 1)?

Solution: When the FSM receives '101', it outputs 1.

2 (d) Draw a FSM that outputs 1 when it receives two or more successive ‘0’.

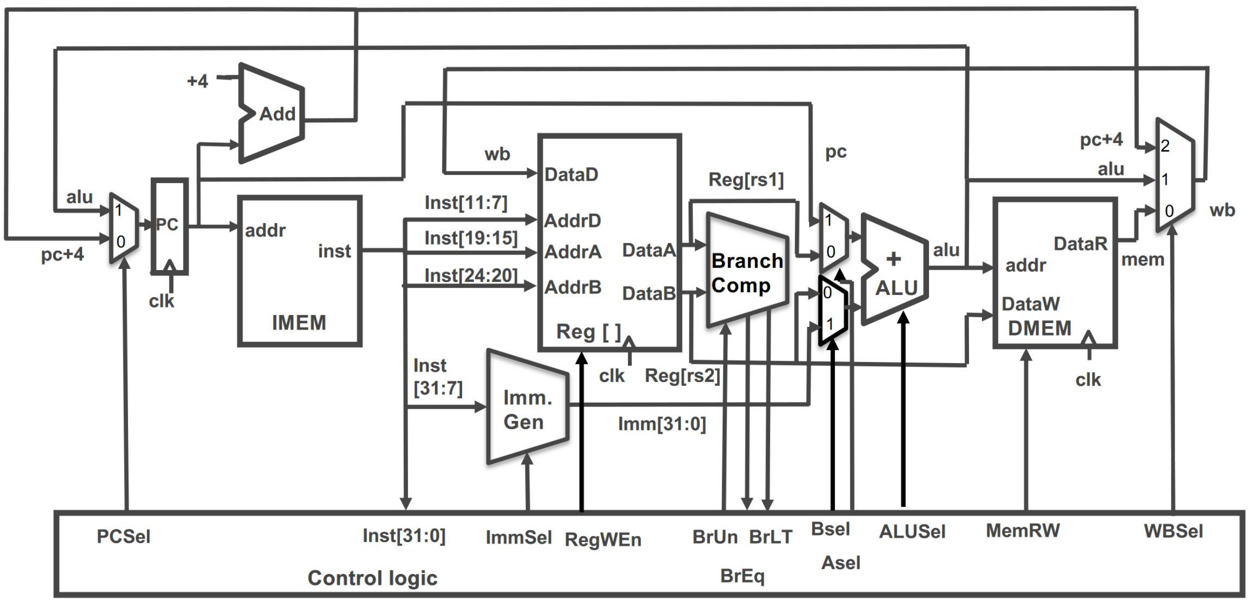


Solution:



12. **RISC-V Datapath**

Here is the datapath we learnt from class:



2 (a) Assume our single-cycle CPU works in 1Ghz, fill in the two blanks.

Stage	IF		EXE	MEM	WB
Time Cost(ps)	200		350	170	130

Solution: ID 150

2 (b) Which of following instructions involves all stages of execution?

A. addi B. jalr C. lw D. auipc

Solution: C

3 (c) Assume t3 = 0x8fffffff, t4 = 0x0fffffff. Write down control signals for **blt t3, t4, label**. Please use * to indicate that what this signal is does not matter.

PCSel	ImmSel	RegWEn	BrUn	BrEq	BrLT	ASel	BSel	ALUSel	MemRW	WBSel

Solution: PCSel = 1 ImmSel = B RegWEn = 0 BrUn = 0 BrEq = 0 BrLT = 1 ASel = 1 BSel = 1 ALUSel = Add MemRW = Read WBSel = *