一、思考题

要求：下面的子问题需要书面完成，可以手写后扫描、也可以直接编辑本文件回答问题。

1、假定某个程序的代码可并行化部分是98%，提供给你多个处理器核(core)来将问题并行化。如果并行化后加速比要达到7以上，至少需要多少处理器核将问题并行化处理。

答案： 1/(0.02 + 0.98/x) >= 7 x>= 7.9 取 8

2、以下是两段 C 语言代码，函数 arith( )是直接用 C 语言写的，而 optarith( )是对 arith( )函数以某个确定的 M 和 N 编译生成的机器代码反编译生成的。根据 optarith( )，可以推断函数 arith( ) 中 M 和 N 的值各是多少？

```
#define    M
#define    N
int    arith      (int x, int y)
{
       int    result = 0 ;
       result = x*M + y/N;
       return result;
}

int optarith ( int x,    int y)
{
       int    t = x;
       x <<= 2;
       x -= t;
       if ( y < 0 )    y += 15;
       y>>4;
       return x+y;
```

参考答案：

可以看出 x*M 和“int t = x; x <<= 2; x-=t;”三句对应，这些语句实现了 x 乘 4 的功能（左移 2 位相当于乘以 4，然后再减 1），因此，M 等于 3；

y/N 与“if (y < 0) y += 15; y>>4;”两句对应，功能主要由第二句“y 右移 4 位”实现，它实现了 y 除以 16 的功能，因此 N=16。而第一句“if (y < 0) y += 15;”主要用于对 y 为负数时进行调整。

3、假设我们在对有符号值使用补码运算的 32 位机器上运行代码。对于有符号值使用的是算术右移，对无符号值使用的是逻辑右移。变量的声明和初始化如下：

```
int x = foo(); //调用某某函数，给 x 赋值
int y = bar(); //调用某某函数，给 y 赋值
unsigned ux = x;
unsigned uy = y;
```

对于下面每个 C 表达式

证明对于所有的 x 和 y 值，都为真（等于 1）；或者（2）给出使得它为假的 x 和 y 值；

A. (x>0) || (x-1<0)

B. (x * x) >= 0

C. x<0 || -x<=0

D. x>0 || -x>=0

E. x+y == uy+ux

F. x*~y + uy*ux == -x

G. x*4 + y*8 == (x<<2)+(y<<3)

H. ((x>>2)<<2)<=x

A 假。设 x 等于-2147483 648（TMin32），那么，我们有 x-1 等于 2147483647（TMax32）

B 假。当 x 为 65535（0xFFFF）时，x*x 为-131071（0xFFFE0001）

C 真。如果 x 是非负数，则-x 是非正的

D 假。设 x 为-2 147 483 648（TMin32），那么 x 和-x 都为负数

E 真。补码和无符号数乘法有相同的位级行为，而且他们是可交换的

F 真。~y 等于-y-1。uy*uy 等于 x*y，因此等式左边等价于 x*-y-x+x*y

G 真。算术左移运算与乘法运算等价

H 真。因为右移总是向负无穷大方向取整

4、给定：

- int 、unsigned int 长度为 32 bits.
- float 是 32 位 IEEE 754 单精度浮点数，double 是 64 位 IEEE 754 双精度浮点数
- 变量之间的装换如下：

```
/* Create some arbitrary values */
int x = random();
int y = random();
int z = random();
 /* Convert to other forms */
unsigned ux = (unsigned) x;
unsigned uy = (unsigned) y;
double dx = (double) x;
double dy = (double) y;
double dz = (double) z;
```

以下表达式，哪些恒定为 true？是的话圈 “Y”，不是的话圈“N” ，并指出原因，给出反例。

Expression	Always True?
`(x<y) == (-x>-y)`	Y N
`((x+y)<<4) + y-x == 17*y+15*x`	Y N
`~x+~y+1 == ~(x+y)`	Y N
`ux-uy == -(y-x)`	Y N
`(x >= 0) || (x < ux)`	Y N
`((x >> 1) << 1) <= x`	Y N
`(double)(float) x == (double) x`	Y N
`dx + dy == (double) (y+x)`	Y N
`dx + dy + dz == dz + dy + dx`	Y N
`dx * dy * dz == dz * dy * dx`	Y N

Answers:

Expression	Always True?
1. (x<y) == (-x>-y)	No: Let x = Tmin, y = 0
2. ((x+y)<<4) + y-x == 17*y+15*x	Yes: Associative, commutative, distributes
3. ~x+~y+1 == ~(x+y)	Yes: (-x-1)+(-y-1)+1 == -(x+y)-1
4. ux-uy == -(y-x)	Yes:
5. (x >= 0) || (x < ux)	No: x = -1. Comparison x < ux is never true.
6. ((x >> 1) << 1) <= x	Yes: x>>1 rounds toward minus infinity.
7. (double)(float) x == (double) x	No: Try x = Tmax.
8. dx + dy == (double) (y+x)	No: Try x=y=Tmin.
9. dx + dy + dz == dz + dy + dx	Yes: Within range of exact representation by double's.
10. dx * dy * dz == dz * dy * dx	No: Try dx=Tmax, dy=Tmax-1, dz=Tmax-2

二、 实践题：位级运算、数的编码

要求：不需要提交代码，只需要在报告中，把你的运行结果、以及你实现的五个函数的源代码贴上来。

先下载 datalab.tar 文件

#解压缩：

$ tar -xvf datalab.tar

#进入文件夹

$ cd datalab

查看文件夹下的文件

$ ls

执行命令

$ make clean

$ make all

运行：

$ ./fshow 一个浮点数例如 34.5

你可以看到这个数的 float 机器编码

$ ./ishow 一个整数例如 20

你可以看到这个数的十六进制的机器编码

$ ./btest

你会看到程序错误的提示：

```
Score   Rating  Errors  Function
ERROR: Test allOddBits(-2147483648[0x80000000]) failed...
...Gives 2[0x2]. Should be 0[0x0]
ERROR: Test isLessOrEqual(-2147483648[0x80000000],-2147483648[0x80000000]) faile
d...
...Gives 2[0x2]. Should be 1[0x1]
ERROR: Test logicalNeg(-2147483648[0x80000000]) failed...
...Gives 2[0x2]. Should be 0[0x0]
ERROR: Test floatScale2(0[0x0]) failed...
...Gives 2[0x2]. Should be 0[0x0]
ERROR: Test floatFloat2Int(0[0x0]) failed...
...Gives 2[0x2]. Should be 0[0x0]
Total points: 0/20
```

你需要做的是：修改 bit.c 文件中的几个函数，完成规定的功能，仔细阅读 bit.c 中各函数前的注释，了解各函数应该能达到的功能。这些函数有：

```
// 判断整数 x 的所有奇数位是否都为 1
//可以使用的运算符：~ & ^ | + << >>
int allOddBits(int x) {
  int mask = 0xAA+(0xAA<<8);
  mask=mask+(mask<<16);
  return !((mask&x)^mask);
}

//使用位级运算符实现判断整数 x<= y
//可以使用的运算符：~ & ^ | + << >>
int isLessOrEqual(int x, int y) {
 int sign = 1 << 31;
 int signx = !(x & sign); // positive or zero is 1, negetive is 0
 int signy = !(y & sign);
 int diff = y + ~x + 1; // diff = y - x
 int sameSign = !(signx ^ signy);
 int lessEq = sameSign & !(diff & sign); // 符号相等 且 y-x >= 0
  return (!signx & signy) | lessEq;
}

//使用位级运算求逻辑非 !
//可以使用的运算符：~ & ^ | + << >>
int logicalNeg(int x) {
  return ((x|(~x+1))>>31)+1;
}
//求 2 乘一个浮点数，
// 可使用任意整数的合法运算符，例如: &, |, ^ , ||, &&, +， -， if, while
unsigned floatScale2(unsigned uf) {
  int sign = uf & 0x80000000;
  int exp = uf & 0x7f800000;
  int frag = uf & 0x007fffff;
  if (exp == 0) {
    //非规格化的数
    return sign | frag << 1;
  }
  if (exp == 0x7f800000) { // inf or NaN
    return uf;
  }
  // 规格化的数
  exp += 0x0800000; // 指数加 1，相当于 乘二
```

```
    if (exp == 0x7f800000) {
        // inf
        frag = 0;
    }
    return sign | exp | frag;
}

// 将浮点数 uf 转换为整数的，返回其 32 位的位级表达
// 可使用任意整数的合法运算符，例如: &, |, ^ , ||, &&, +，-，  if, while
int floatFloat2Int(unsigned uf)
 {
    int sign = uf & 0x80000000;
    int exp = ((uf & 0x7f800000) >> 23) - 127; // 规格化数的指数的真值（非规格化数一律返回0）
    int frag = (uf & 0x007fffff) | 0x00800000; // 补上前导 1
    int absval;
    if (exp < 0) {
        return 0;
    }
    if (exp > 30) {
        return 0x80000000;
    }
    if (exp < 23) {
        // 需要截断部分尾数
        absval = frag >> (23 - exp);
    } else {
        absval = frag << (exp - 23);
    }
    return sign == 0x80000000 ? -absval : absval;
 }
```

关于浮点数的编码，IEEE-754 Floating Point Converter (h-schmidt.net) 这里有一个转换器，希望对你有帮助。

修改程序后重新编译：

$ make clean

$ make all

执行：

$ ./btest

如果你的实现全部正确，应该得到以下结果，这 5 个函数的得分情况如下，满分为 20 分。

```
Score   Rating  Errors  Function
 2      2       0       allOddBits
 4      4       0       isLessOrEqual
 4      4       0       logicalNeg
 5      5       0       floatScale2
 5      5       0       floatFloat2Int
Total points: 20/20
```

如果有扣分，说明函数实现没有符合要求，使用了不允许使用的运算符。使用命令
$ ./dlc bits.c
可以调用文件包中提供的规则检查器，检查哪个运算符是不合规定的。

注：本实验选自 CMU CSAPP，如果你想了解 CMU CSAPP：Datalab 的完整要求，可以查看：http://csapp.cs.cmu.edu/3e/labs.html 找到相关的文档和代码。

RISC-V 汇编语言 作业与实验

目标

- 熟悉使用 Venus 模拟器
- 练习运行和调试 RISC-V 汇编代码。
- 编写 RISC-V 函数。
- 理解嵌套函数调用、指针、数组、链表在汇编语言中的实现
- 理解变址寻址方式的多种用途

RISC-V 模拟器简介

汇编语言是接近机器代码的低级语言。因为您的计算机只能运行（x86 或 ARM）的机器代码，无法直接在您的机器上直接执行 RISC-V 代码。所以使用 RISC-V 模拟器 Venus。你可以通过查看 Venus 参考手册了解它的使用方法。

练习 1：熟悉 Venus

用任何一个编辑器打开 ex1.s 文件， 观察文件内容我们会发现：标号后面有一个冒号(:)， 注释以井号(#)开始，每行只能写一条指令。程序最开始部分是 main 函数的语句；程序结束（exit）时，将参数值设置为 10，并调用 ecall 指令。 通过设置不同的参数并调用 ecall，可以实现各种系统调用。例如：输出一些内容到显示器终端、读文件、写文件、创建子进程等，都是系统调用.

- 直接将 ex1.s 从本地计算机复制/粘贴到 Venus 编辑器中。
- 单击“Simulator”选项卡，然后单击“Assemble & Simulate from Editor”按钮。模拟器就准备好要执行的代码了。如果单击“Editor”选项卡，模拟将重置。
- 在模拟器中，要执行下一条指令，请单击“step”按钮。
- 要撤消指令，请单击“prev”按钮。
- 要运行程序直至完成，请单击“run”按钮。
- 要重置程序，请单击“reset”按钮。
- 所有 32 个寄存器的内容都在右侧，控制台输出在底部。
- 要查看内存内容，请单击右侧的“Memory”选项卡。您可以使用底部的下拉菜单导航到内存的不同部分。

思考以下问题：

1) .data, .word, .text 指令的含义是什么？（即：它们的用途是什么？）
2) 运行程序直到完成。程序输出了什么数字？如果 0 是第 0 个斐波那契数，那么这是第几个斐波那契数？
3) n 存储在内存中的哪个地址？提示：查看寄存器的内容。
4) 在不修改代码（不改变“Edit”栏下的代码）的情况下，手动修改寄存器的值来计算第 13 个斐波那契数（索引从 0 开始）。应该修改的寄存器是哪个？

练习 2：从 C 翻译到 RISC-V

打开文件 ex2.c 和 ex2.s。汇编代码（ex2.s）是给定 C 程序（ex2.c）到 RISC-V 的汇编翻译。

阅读汇编代码，思考以下问题：

1) 表示变量 k 的寄存器？
2) 表示总和变量 sum 的寄存器；
3) 分别指向源数组和目标数组的指针的寄存器；
4) 在汇编指令里，指针怎么表示？

练习 3：用 map 调用 RISC-V 函数
本练习将使用文件 list_map.s 中的汇编语言程序。
在 C 语言中，链表中的一个结点的数据类型被定义为：

```
struct node {
    int value;
    struct node *next;
};
```

为了实现一个 map 函数，其功能是：递归地遍历链表，将指定函数应用于链表的每一个结点的 value，并将返回的值存储在相应的节点中。在 C 语言中，map 函数的定义是这样的：

```
void map(struct node *head, int (*f)(int))
{     //f 是函数指针，指向某个函数的起始地址， 这个函数只有一个参数，该参数类型为 int
    if (!head) { return; }
    head->value = f(head->value);
    // 提示，转换为 RISC-V 汇编调用函数 f 时，应该使用 JALR, 而不是 JAL
    map(head->next,f);
}
```

如果你不了解“函数指针”，建议先补充这方面的知识。函数指针是一个指向函数的指针变量，其本质是一个指针，代表函数的内存地址。

例如：有一个函数 square:
int square(int i) { return i*i; }
map 的第一个参数是一个值为 32 位整数的单链表的头节点的地址。
square 可以作为第二个参数传递给 map。

在本练习中，我们将在 RISC-V 汇编程序中完成 list map 的实现。函数的实现过程中，是对链表原位置中的值进行改变，而不是创建并返回带有修改值的新链表。

补充 list_map.s 中的汇编语言程序，使得其输出为：

```
9 8 7 6 5 4 3 2 1 0
81 64 49 36 25 16 9 4 1 0
```

第一行是原始链表中每一个结点的值，第二行是应用 map 的第二个参数指定的函数（本例为：square）对每一个结点修改后、链表中各个结点的值。
添加您的代码，确保运行的结果为上述结果。

要求：在实验报告中，把运行结果、以及你实现的函数的源代码（把你添加的部分高亮显示）贴上来。

练习 4：
本练习和练习 3 稍微有些不同， 在本练习中，链表中的一个结点的数据类型被定义为：

```
struct node {
    int *arr;
    int size;
    struct node *next;
};
```

链表中每一个节点是数组，arr 是数组的地址，size 是数组的大小。
新的 map 函数的作用是：遍历链表中的每一个节点中的数组，将函数 f 运用于数组中的每一个元素，结果写回数组中对应位置。

```
void map(struct node *head, int (*f)(int)) {
    if (!head) { return; }
    for (int i = 0; i < head->size; i++) {
      head->arr[i] = f(head->arr[i]);
    }
    map(head->next, f);
}
```

给定的文件 megalistmanips.s， 其中函数 f（int x）的功能是计算并返回 x*(x+1)，本次 map 的正确运行结果应该如下：

```
Lists before:
5 2 7 8 1
1 6 3 8 4
5 2 7 4 3
1 2 3 4 7
5 6 7 8 9

Lists after:
30 6 56 72 2
2 42 12 72 20
30 6 56 20 12
2 6 12 20 56
30 42 56 72 90
```

但 megalistmanips.s 这个文件存在错误，没有正确运行，请找出它的错误，并修正。
一些提示：

- jal 进行函数调用之前，我们需要将哪些内容入栈？
- add t0, s0, x0 和 lw t0, 0(s0) 这两条指令区别是什么？
- 注意正确表示结构体 node 中的成员所属的数据类型
- 重点修改 map, mapLoop 部分， 其余函数，例如 done 不用修改，但它可以帮助理解整个程序.

- 除了 s0 和 s1， 不允许使用另外的 s 开头的寄存器（save register: s2-s11 不能再使用）,你可以使用 temporary register(例如 t1, t2 等)，并遵循约定的寄存器使用规范。

要求：在实验报告中，把运行结果、以及你实现的函数的源代码（把你修改的部分高亮显示）贴上来。

练习 5：写一个没有条件分支（branch）的函数

有一个 （discrete valued function）离散值函数 f ，作用于一个整数集合 {-3, -2, -1, 0, 1, 2, 3}. 函数的定义如下：

```
f(-3) = 6
f(-2) = 61
f(-1) = 17
f(0) = -38
f(1) = 19
f(2) = 42
f(3) = 5
```

在 `discrete_fn.s` 文件中实现该函数，注意：**不能使用任何 branch 或者 jump 指令**

要求：在实验报告中，把运行结果、以及你实现的函数的源代码（把你添加的部分高亮显示）贴上来。

注：本作业选自 UC Berkeley 大学 CS61C 课程 Lab3 和 lab4，相关链接：
https://inst.eecs.berkeley.edu/~cs61c/su20/labs/lab03/
https://inst.eecs.berkeley.edu/~cs61c/su20/labs/lab04/

练习 3 参考答案：

```
.globl map

.text
main:
    jal ra, create_default_list
    add s0, a0, x0  # a0 = s0 is head of node list

    #print the list
    add a0, s0, x0
    jal ra, print_list

    # print a newline
```

```
    jal ra, print_newline

    # load your args
    add a0, s0, x0  # load the address of the first node into a0

    # load the address of the function in question into a1 (check out la on the
green sheet)
    ### YOUR CODE HERE ###
    la a1, square

    # issue the call to map
    jal ra, map

    # print the list
    add a0, s0, x0
    jal ra, print_list

    # print another newline
    jal ra, print_newline

    addi a0, x0, 10
    ecall #Terminate the program

map:
    # Prologue: Make space on the stack and back-up registers
    ### YOUR CODE HERE ###
    addi sp, sp, -8
    sw ra, 0(sp)
    sw s0, 4(sp)

    beq a0, x0, done    # If we were given a null pointer (address 0), we're
done.

    add s0, a0, x0  # Save address of this node in s0
    add s1, a1, x0  # Save address of function in s1

    # Remember that each node is 8 bytes long: 4 for the value followed by 4
for the pointer to next.
    # What does this tell you about how you access the value and how you access
the pointer to next?

    # load the value of the current node into a0
    # THINK: why a0?
    ### YOUR CODE HERE ###
```

```
    lw a0, 0(s0)

    # Call the function in question on that value. DO NOT use a label (be
prepared to answer why).
    # What function? Recall the parameters of "map"
    ### YOUR CODE HERE ###
    jalr s1

    # store the returned value back into the node
    # Where can you assume the returned value is?
    ### YOUR CODE HERE ###
    sw a0, 0(s0)

    # Load the address of the next node into a0
    # The Address of the next node is an attribute of the current node.
    # Think about how structs are organized in memory.
    ### YOUR CODE HERE ###
    lw a0, 4(s0)

    # Put the address of the function back into a1 to prepare for the recursion
    # THINK: why a1? What about a0?
    ### YOUR CODE HERE ###
    addi a1, s1, 0

    # recurse
    ### YOUR CODE HERE ###
    jal map

done:
    # Epilogue: Restore register values and free space from the stack
    ### YOUR CODE HERE ###
    lw ra, 0(sp)
    lw s0, 4(sp)
    addi sp, sp, 8

    jr ra # Return to caller

square:
    mul a0 ,a0, a0
    jr ra

create_default_list:
    addi sp, sp, -12
    sw  ra, 0(sp)
```

```
    sw  s0, 4(sp)
    sw  s1, 8(sp)
    li  s0, 0       # pointer to the last node we handled
    li  s1, 0       # number of nodes handled
loop:   #do...
    li  a0, 8
    jal ra, malloc      # get memory for the next node
    sw  s1, 0(a0)   # node->value = i
    sw  s0, 4(a0)   # node->next = last
    add s0, a0, x0  # last = node
    addi    s1, s1, 1   # i++
    addi t0, x0, 10
    bne s1, t0, loop    # ... while i!= 10
    lw  ra, 0(sp)
    lw  s0, 4(sp)
    lw  s1, 8(sp)
    addi sp, sp, 12
    jr ra

print_list:
    bne a0, x0, printMeAndRecurse
    jr ra       # nothing to print
printMeAndRecurse:
    add t0, a0, x0  # t0 gets current node address
    lw  a1, 0(t0)   # a1 gets value in current node
    addi a0, x0, 1      # prepare for print integer ecall
    ecall
    addi    a1, x0, ' '     # a0 gets address of string containing space
    addi    a0, x0, 11      # prepare for print string syscall
    ecall
    lw  a0, 4(t0)   # a0 gets address of next node
    jal x0, print_list  # recurse. We don't have to use jal because we already
have where we want to return to in ra

print_newline:
    addi    a1, x0, '\n' # Load in ascii code for newline
    addi    a0, x0, 11
    ecall
    jr  ra

malloc:
    addi    a1, a0, 0
    addi    a0, x0 9
    ecall
```

```
    jr  ra
```

练习 4 参考答案：

```
.globl map

.data
arrays: .word 5, 6, 7, 8, 9
        .word 1, 2, 3, 4, 7
        .word 5, 2, 7, 4, 3
        .word 1, 6, 3, 8, 4
        .word 5, 2, 7, 8, 1

start_msg:  .asciiz "Lists before: \n"
end_msg:    .asciiz "Lists after: \n"

.text
main:
    jal create_default_list
    mv s0, a0   # v0 = s0 is head of node list

    #print "lists before: "
    la a1, start_msg
    li a0, 4
    ecall

    #print the list
    add a0, s0, x0
    jal print_list

    # print a newline
    jal print_newline

    # issue the map call
    add a0, s0, x0      # load the address of the first node into a0
    la  a1, mystery     # load the address of the function into a1

    jal map

    # print "lists after: "
    la a1, end_msg
    li a0, 4
```

```
    ecall

    # print the list
    add a0, s0, x0
    jal print_list

    li a0, 10
    ecall

map:
    addi sp, sp, -12
    sw ra, 0(sp)
    sw s1, 4(sp)
    sw s0, 8(sp)

    beq a0, x0, done    # if we were given a null pointer, we're done.

    add s0, a0, x0      # save address of this node in s0
    add s1, a1, x0      # save address of function in s1
    add t0, x0, x0      # t0 is a counter

    # remember that each node is 12 bytes long:
    # - 4 for the array pointer
    # - 4 for the size of the array
    # - 4 more for the pointer to the next node
mapLoop:
    lw t1, 0(s0)     # load the address of the array of current node into
t1
    lw t2, 4(s0)        # load the size of the node's array into t2

    li t3, 4
    mul t3, t0, t3
    add t1, t1, t3      # offset the array address by the count
    lw a0, 0(t1)        # load the value at that address into a0

    addi sp, sp, -12
    sw ra, 0(sp)
    sw t0, 4(sp)
    sw t1, 8(sp)
    jalr s1             # call the function on that value.
    lw ra, 0(sp)
    lw t0, 4(sp)
    lw t1, 8(sp)
    addi sp, sp, 12
```

```
    sw a0, 0(t1)        # store the returned value back into the array
    addi t0, t0, 1      # increment the count
    bne t0, t2, mapLoop # repeat if we haven't reached the array size yet

    lw a0, 8(s0)        # load the address of the next node into a0
    mv a1, s1           # put the address of the function back into a1 to
prepare for the recursion

    jal  map            # recurse
done:
    lw s0, 8(sp)
    lw s1, 4(sp)
    lw ra, 0(sp)
    addi sp, sp, 12
    jr ra

mystery:
    mul t1, a0, a0
    add a0, t1, a0
    jr ra

create_default_list:
    addi sp, sp, -4
    sw ra, 0(sp)
    li s0, 0  # pointer to the last node we handled
    li s1, 0  # number of nodes handled
    li s2, 5  # size
    la s3, arrays
loop: #do...
    li a0, 12
    jal malloc      # get memory for the next node
    mv s4, a0
    li a0, 20
    jal  malloc     # get memory for this array

    sw a0, 0(s4)    # node->arr = malloc
    lw a0, 0(s4)
    mv a1, s3
    jal fillArray   # copy ints over to node->arr

    sw s2, 4(s4)    # node->size = size (4)
    sw  s0, 8(s4)   # node-> next = previously created node
```

```
    add s0, x0, s4  # last = node
    addi s1, s1, 1  # i++
    addi s3, s3, 20 # s3 points at next set of ints
    li t6 5
    bne s1, t6, loop # ... while i!= 5
    mv a0, s4
    lw ra, 0(sp)
    addi sp, sp, 4
    jr ra

fillArray: lw t0, 0(a1) #t0 gets array element
    sw t0, 0(a0) #node->arr gets array element
    lw t0, 4(a1)
    sw t0, 4(a0)
    lw t0, 8(a1)
    sw t0, 8(a0)
    lw t0, 12(a1)
    sw t0, 12(a0)
    lw t0, 16(a1)
    sw t0, 16(a0)
    jr ra

print_list:
    bne a0, x0, printMeAndRecurse
    jr ra   # nothing to print
printMeAndRecurse:
    mv t0, a0 # t0 gets address of current node
    lw t3, 0(a0) # t3 gets array of current node
    li t1, 0  # t1 is index into array
printLoop:
    slli t2, t1, 2
    add t4, t3, t2
    lw a1, 0(t4) # a0 gets value in current node's array at index t1
    li a0, 1  # preparte for print integer ecall
    ecall
    li a1, ' ' # a0 gets address of string containing space
    li a0, 11  # prepare for print string ecall
    ecall
    addi t1, t1, 1
  li t6 5
    bne t1, t6, printLoop # ... while i!= 5
    li a1, '\n'
    li a0, 11
    ecall
```

```
    lw a0, 8(t0) # a0 gets address of next node
    j print_list # recurse. We don't have to use jal because we already have where we want to return to in ra

print_newline:
    li a1, '\n'
    li a0, 11
    ecall
    jr ra

malloc:
    mv a1, a0 # Move a0 into a1 so that we can do the syscall correctly
    li a0, 9
    ecall
    jr ra
```

练习5参考答案

```
.globl f

.data
neg3:   .asciiz "f(-3) should be 6, and it is: "
neg2:   .asciiz "f(-2) should be 61, and it is: "
neg1:   .asciiz "f(-1) should be 17, and it is: "
zero:   .asciiz "f(0) should be -38, and it is: "
pos1:   .asciiz "f(1) should be 19, and it is: "
pos2:   .asciiz "f(2) should be 42, and it is: "
pos3:   .asciiz "f(3) should be 5, and it is: "

output: .word   6, 61, 17, -38, 19, 42, 5
.text
main:
    la a0, neg3
    jal print_str
    li a0, -3
    la a1, output
    jal f               # evaluate f(-3); should be 6
    jal print_int
    jal print_newline

    la a0, neg2
    jal print_str
    li a0, -2
    la a1, output
    jal f               # evaluate f(-2); should be 61
```

```
jal print_int
jal print_newline

la a0, neg1
jal print_str
li a0, -1
la a1, output
jal f               # evaluate f(-1); should be 17
jal print_int
jal print_newline

la a0, zero
jal print_str
li a0, 0
la a1, output
jal f               # evaluate f(0); should be -38
jal print_int
jal print_newline

la a0, pos1
jal print_str
li a0, 1
la a1, output
jal f               # evaluate f(1); should be 19
jal print_int
jal print_newline

la a0, pos2
jal print_str
li a0, 2
la a1, output
jal f               # evaluate f(2); should be 42
jal print_int
jal print_newline

la a0, pos3
jal print_str
li a0, 3
la a1, output
jal f               # evaluate f(3); should be 5
jal print_int
jal print_newline

li a0, 10
```

```
    ecall

# f takes in two arguments:
# a0 is the value we want to evaluate f at
# a1 is the address of the "output" array (defined above).
# Think: why might having a1 be useful?
f:
    # YOUR CODE GOES HERE!
    addi t0, a0, 3
    li t1, 4
    mul t0, t0, t1
    add t0, a1, t0
    lw a0, 0(t0)
    jr ra               # Always remember to jr ra after your function!

print_int:
    mv a1, a0
    li a0, 1
    ecall
    jr    ra

print_str:
    mv a1, a0
    li a0, 4
    ecall
    jr    ra

print_newline:
    li a1, '\n'
    li a0, 11
    ecall
    jr    ra
```

作业 3： RISC-V 处理器

1. 下图是一个实现了 RV32I 指令集的单周期处理器的数据通路，为使得不同的指令在同一个处理器中完成不同的功能，控制器对指令译码后，要发出不同的控制信号。表格中罗列了部分指令的控制信号。其中 * 表示控制信号可以任意取值；BrUn 为 1 时，表示比较指令的两个比较值是无符号数；Immsel 取不同的值，是因为不同的指令中的立即数在指令中不同的位置、以及立即数扩展的方法不同。

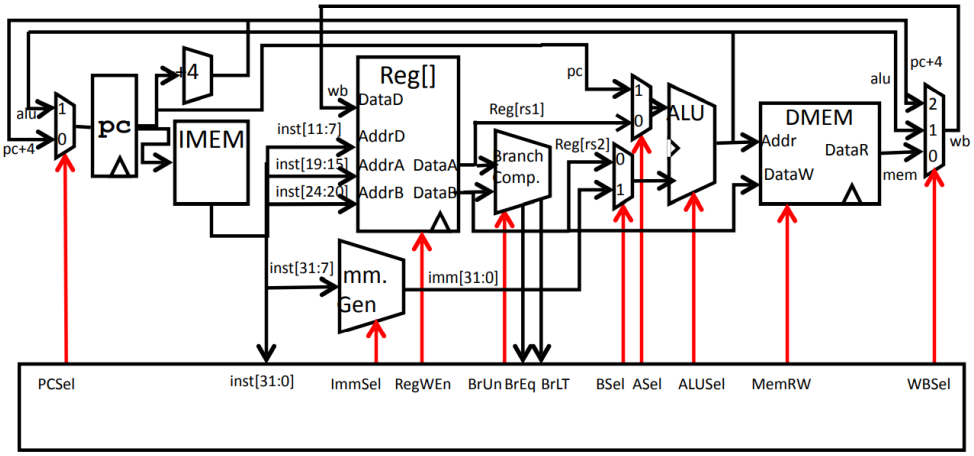


图 1. RV32I 单周期处理器数据通路

	BrEq	BrLT	PCSel	ImmSel	BrUn	ASel	BSel	ALUSel	MemRW	RegWEn	WBSel
add	*	*	0 (PC + 4)	*	*	0 (Reg)	0 (Reg)	add	0	1	1 (ALU)
ori	*	*	0	I	*	0 (Reg)	1 (Imm)	or	0	1	1 (ALU)
lw	*	*	0	I	*	0 (Reg)	1 (Imm)	add	0	1	0(MEM)
sw	*	*	0	S	*	0 (Reg)	1 (Imm)	add	1	0	*
beq	1/0	*	1/0	SB	*	1 (PC)	1 (Imm)	add	0	0	*
jal	*	*	1 (ALU)	UJ	*	1 (PC)	1 (Imm)	add	0	1	2(PC + 4)
bltu	*	1/0	1/0	SB	1	1 (PC)	1 (Imm)	add	0	0	*

表 1. 部分指令的控制信号

回答问题：参考上面的图表，分别写出 slli 指令和 AUIPC 指令的控制信号

参考信息如下：

SLLI 用法举例: sllix11,x12,2 表示 x11 = x12<<2（Shift Left Logical Immediate）

SLLI 指令编码格式：

imm			funct3		opcode	
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI

AUIPC 指令用法举例:

```
auipc x1, <hi20bits>  # Adds upper immediate value to
                      #  and places result in x1
```

AUIPC 指令编码格式：

31	12 11	7 6	0
imm[31:12]	rd	opcode	
20	5	7	
U-immediate[31:12]	dest	AUIPC	

更多关于 RISC-V 指令的信息可参考：

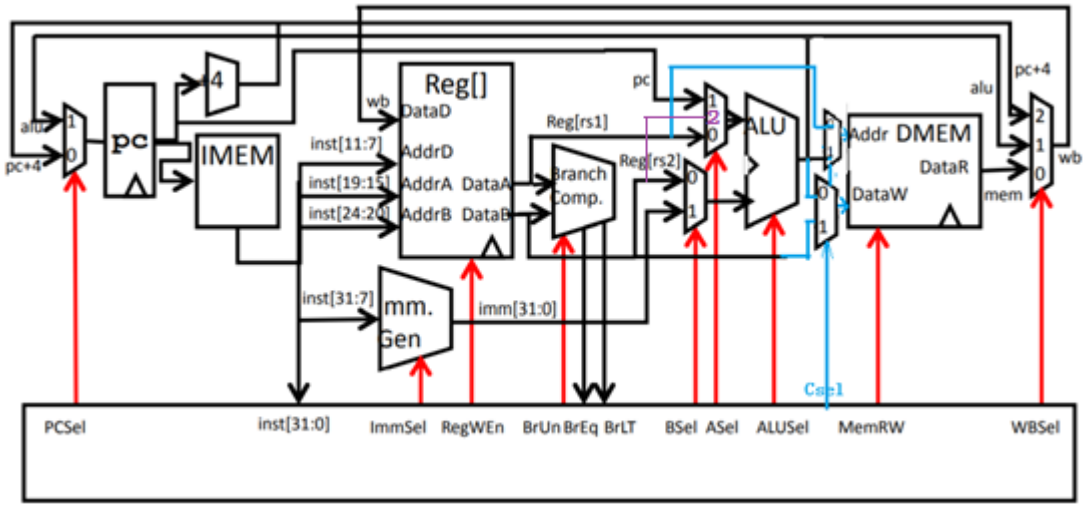
http://inst.eecs.berkeley.edu/~cs61c/resources/RISCV_Green_Sheet.pdf

答案：

	BrEq	BrLT	PCSel	ImmSel	BrUn	ASel	Bsel	ALUSel	MemRW	RegWEn	WBSel
slli	*	*	0(PC+4)	I	*	0	1	sll	read	1	1(ALU)
auipc	*	*	0(PC+4)	U	*	1	1	add	read	1	1(ALU)

2. 尝试添加 RISC-V 指令： ss rs1, rs2, imm
指令的功能为： Mem[Reg[rs1]] = Reg [rs2] + immediate （存储两数之和）

回答问题：为了支持这条指令，对图 1 中的 RV32I 单周期处理器，需要添加的新功能部件是什么？现有的哪些部件需要改造？需要新添加的数据通路是什么？ 为控制单元新添加的控制信号有哪些？



答案：
本题没有标准答案，取决于你怎么编码、以及数据通路的设计。 以下是 RISC-V 中 S 型指令的编码：

31 25	24 20	19 15	14 12	11 7	6 0	
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	S sb
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	S sh
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	S sw

如果采用类似的 pattern 对 SS 指令进行编码，rs1 是第 15-19 位， rs2 是第 20-24 位，从 ALU 输出的 DataA 是 Reg[rs1]的值，DataB 是 Reg[rs2] 的值。
数据通路可以修改为：

1) ALU 的一个输入端从两输入改为三输入，Asel 控制信号升级为两位，Asel=2 时(如图中紫色线体部分)， 它的输入端是 **Reg[rs2] 的值。**
2) 需要增加两个 MUX 分别在 DMEM 的 Addr 输入端门口， 和 DMEM 的 DataW 输入端门口。使用一个信号 Csel 同时控制这两个 MUX. 如上图增加的蓝色线体部分示意。

以下是改造后的数据通路，实现不同指令时的控制信号。

- 当 Csel=0(addr=reg1, dataw=ALUouput)、Asel=2(reg2)、Bsel=1(imm)、MemRW=write、RegWEn=0、Pcsel=0、Alusel=Add、BrUn=*、immsel=S、WBsel=*时， ALU 的输出作为要写入内存的数据， Reg[rs1]的值作为写入的内存地址， **实现了新的 ss 指令**。
- 当 Csel=1(addr=ALUoput , dataw=Reg2) 、Asel=0(reg1)、Bsel=1 (imm)、MemRW=write、

RegWEn=0、Pcsel=0、Alusel=Add、BrUn=*、immsel=S、 WBsel=*时， Reg[rs2]的值作为要写入内存的数据， Reg[rs1]+imm 的运算结果，也就是 ALU 的输出、作为写入的内存地址，这是一条正常的 sw 指令。

- 当 Csel=1、Asel=0(reg1)、Bsel=1(imm)、MemRW=Read、RegWEn=1、Pcsel=0、Alusel=Add、BrUn=*、immsel=I、WBsel=Mem 时，时，Reg[rs1]+imm 的运算结果，也就是 ALU 的输出、作为读的内存地址，从内存中读出的内容，写入 rd(指令的第 7-11w 位指定的寄存器)， 这是一条正常的 lw 指令。

综述，改变后的数据通路，对原有的指令的实现基本上没有改变，仅在实现其他指令时，设置 Csel=1，其余控制信号的取值保持原来的设置。

3 单周期处理器的性能分析

时钟分析方法：

- 每个状态元件的输入信号必须在时上升沿之前稳定下来。
- 关键路径（critical path）：电路中状态元件之间最长的延迟路径。
- $t_{clk} \geqslant t_{clk\text{-}to\text{-}q} + t_{CL} + t_{setup}$，其中 t_{CL} 是组合逻辑中的关键路径
- 如果我们把寄存器放在关键路径上，我们可以通过减少寄存器之间的逻辑量来缩短周期。

电路元件的延时如下所示：

Element	Register clk-to-q	Register Setup	MUX	ALU	Mem Read	Mem Write	RegFile Read	RegFile Setup
Parameter	$t_{clk\text{-}to\text{-}q}$	t_{setup}	t_{mux}	t_{ALU}	$t_{MEMread}$	$t_{MEMwrite}$	t_{RFread}	$T_{RFsetup}$
Delay(ps)	30	20	25	200	250	200	150	20

关于硬件中的时钟的一些术语说明：

- 时钟（CLK）：使系统同步的稳定方波
- 启动时间（setup time）：在时钟边沿之前，输入必须稳定的时间
- 保持时间（hold time）：在时钟边沿之后，输入必须稳定的时间
- “CLK-to-Q”延迟（“CLK-to-Q” delay）：从时钟边沿测量，改变输出需要多长时间
- 周期（period）=最大延迟=“CLK-to-Q”延迟+CL 延迟+启动时间
- 时钟频率=1/周期（即周期的倒数）

回答问题：

1) 用到关键路径（critical path）的指令是哪一条？

最长关键路径指令是 LW 指令

2) 最小时钟周期 t_{clk}是多少？最大时钟频率 f_{clk}是什么？假设 $t_{clk\text{-}to\text{-}q}$ > 保持时间(hold time).
提示：tclk= PC 寄存器的 clktoQ + 关键路径（critical path）延迟 + RegFile_Setup

tclk ≥ tPC clk-to-q + tIMEM read + tRF read + tmux + tALU + tDMEM read + tmux + tRF setup ≥ 30 + 250 + 150 + 25 + 200 + 250 + 25 + 20 ≥ 950 ps
1/950 ps = 1.05 GHz
Note that the delay of branch comparator is omitted because branch comparison is done in parallel with RegFile/ALU, which takes much longer time

4 流水线处理器设计（Pipelined CPU Design）

现在，我们将使用流水线方法来优化一个单周期处理器。流水线虽然增加了单个任务的延迟，但它可以减少时钟周期，提高吞吐量。 在流水线处理器中，多条指令重叠执行，体现了指令级并行性。

为了设计流水线，我们已经将单周期处理器分成五个阶段，在每两个阶段之间增加流水段寄存器。

接下来进行性能分析：

我们将使用与上一题相同的时钟参数：

Element	Register clk-to-q	Register Setup	MUX	ALU	Mem Read	Mem Write	RegFile Read	RegFile Setup
Parameter	$t_{clk\text{-}to\text{-}q}$	t_{setup}	t_{mux}	t_{ALU}	$t_{MEMread}$	$t_{MEMwrite}$	t_{RFread}	$T_{RFsetup}$
Delay(ps)	30	20	25	200	250	200	150	20

回答问题：

1) 这个五阶段流水线处理器的最小时钟周期长度和最大时钟频率分别是多少？

IF : tPC clk-to-q + tIMEM read + tReg setup = 30 + 250 + 20 = 300 ps
ID : tReg clk-to-q + tRF read + tReg setup = 30 + 150 + 20 = 200 ps
EX : tReg clk-to-q + tmux + tALU + tReg setup + tmux = 30 + 25 + 200 + 20 + 25 = 300 ps
MEM : tReg clk-to-q + tDMEM read + tReg setup = 30 + 250 + 20 = 300 ps
WB : tReg clk-to-q + tmux + tRF setup = 30 + 25 + 20 = 75 ps
max(IF, ID, EX,MEM,WB) = 300 ps

NOTE: For the EX stage, the branch comparator time is overshadowed by the ALU computation (The same would be true in the ID stage as well, but since there is no mentioned time for Immediate Generator, we assumed here it is trivial):
Branch comparator : tPC clk-to-q + tBranch comp. = 30 + 75 = 105 ps
Branch target computation : tReg clk-to-q + tmux + tALU + tmux + tPC setup = 300 ps
ALU computation: tReg clk-to-q + tmux + tALU + tReg setup =275

2) 相比于单周期处理器，性能加速比（speed up）是多少？为什么加速比会小于 5？

950 ps /300 ps=3.2 times speedup.
The speedup is less than 5 because of
(1) the necessity of adding pipeline registers, which have clk-to-q and setup times, and
(2) the need to set the clock to the maximum of the five stages, which take different amounts of time.
Note: because of hazards, which require additional logic to resolve, the actual speedup would likely be even less than 3.2 times.

5 图 2 给出了 RV32I 五阶段流水化处理器示意图：

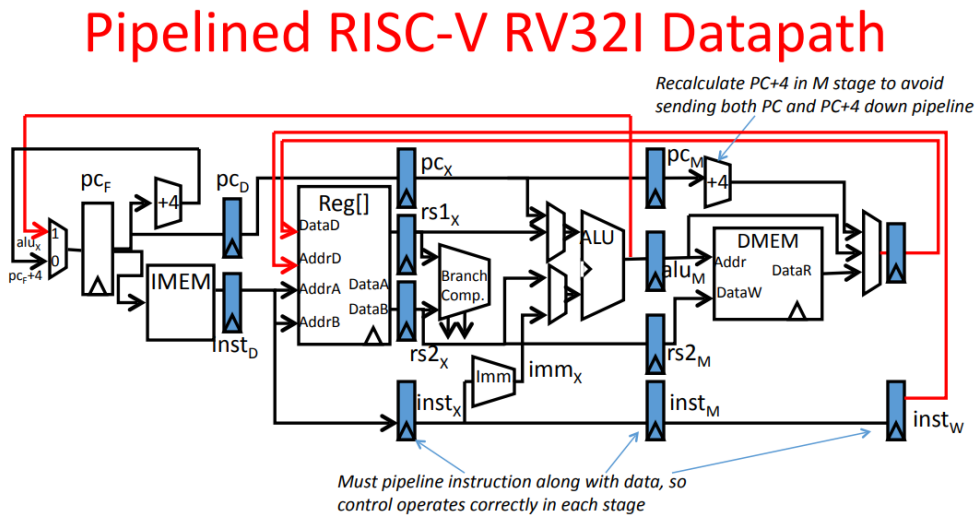


图 2. RV32I 五阶段流水处理器

在该处理器上执行以下指令序列:

```
add  x15,x12,x11
ld    x13, 4(x15)
ld    x12 , 0(x2)
or    x13, x15,x13
sd    x13, 0(x15)
```

注：寄存器堆先写后读，ID 阶段的指令可以在同一个周期内得到 WB 阶段写回的数据;

回答问题：

（1） 假设硬件不检查和处理数据冒险，数据通路没有前向传递（Forwarding）在指令序列中插入空指令 NOP，使得上述指令序列得到正确的执行结果。

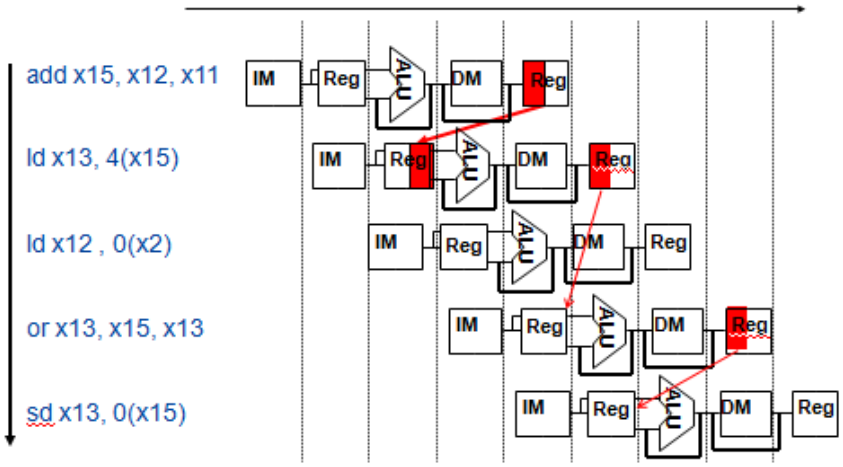
（2） 假设对硬件不做任何改变，进行编译优化：对代码的次序重排、寄存器换名，使得插入的空指令最少。假设寄存器 x17 可以被用来做临时寄存器。

（3） 假设进行硬件优化：数据通路中增加了前向传递（forwarding），并增加了冒险检测单元。 哪些指令之间还是需要停顿？停顿几个周期？

答案：

（1） 各条指令的 数据相关性如下图所表示

由于寄存器堆先写后读，ID 阶段的指令可以在同一个周期内得到 WB 阶段写回的数据。我们需要添加两个 NOP 指令 2 前，使得指令 2 到达 ID 时指令 1 已到达 WB 阶段。同理对于指令 4 和指令 5 前分别应该插入 1 个和 2 个空指令。

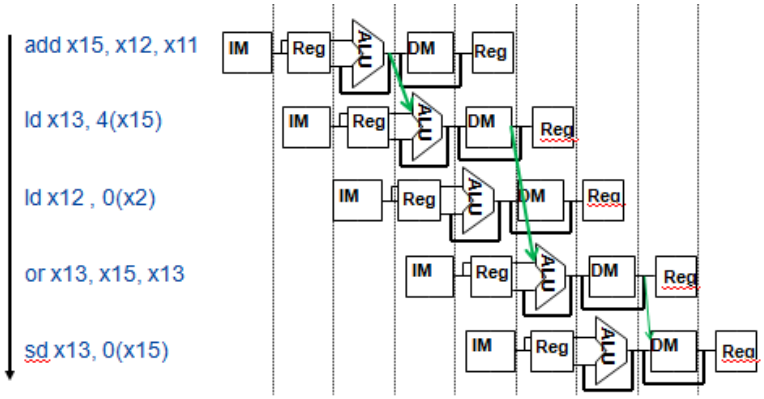


❑ 写后读数据冒险（ Read before write data hazard ）

add x15, x12, x11
NOP
NOP
ld x13, 4(x15)
ld x12 , 0(x2)
NOP
or x13, x15, x13
NOP
NOP
sd x13, 0(x15)

（2） 次序重排和寄存器换名都无法优化

（3） 如下图所示，**如果在 ALU 、DM 输入端都有 forwarding,** 没有指令需要停顿



add x15, x12, x11
ld x13, 4(x15)
ld x12 , 0(x2)
or x13, x15, x13
sd x13, 0(x15)

6 在一个采用“取指、译码/取数、执行、访存、写回”的五段流水线中，检测结果是否为“零”的操作在执行阶段进行，则因为分支延迟导致阻塞的时钟周期为多少？以下指令序列中，在有数据转发（forwarding）的情况下，哪些指令执行时会发生流水线阻塞？各需要阻塞几个时钟周期？

```
1   loop:    add   t1, s3, s3
2            add   t1, t1, t1
3            add   t1, t1, s6
4            lw    t0, 0(t1)
5            bne   t0, s5, exit
6            add   s3, s3, s4
7            j     loop    //这是一条伪指令，是 jal 指令的变体
8   exit:
```

答案：此处，第 2、3、4 条指令所需的操作数可通过“转发”得到，无需加 nop 指令。第 5 条 bne 指令所需的操作数$t0 是 load-use 冒险，不能用“转发”解决问题，需要在第 5 条指令前加一条 nop 指令，或通过硬件将第 5 条指令的执行阻塞一个时钟周期。

j 指令如果在**译码阶段**就根据译码结果计算跳转目标地址，那么 j 指令后面指令会被阻塞 1 个时钟周期，若在**执行阶段**计算，则要阻塞 2 个时钟周期。

作业（存储器、高速缓存、虚拟内存）

一、存储器

1. 存储器容量为 64MB，字长 64 位，体（bank）数 m = 8，分别用顺序方式和交叉方式进行组织。存储周期 T = 100ns,数据总线宽度为 64 位，总线周期 τ = 10ns .问顺序存储器和交叉存储器的带宽（单位：位/秒）各是多少？

解：信息总量： q = 64 位 ×8 =512 位

顺序存储器和交叉存储器读出 8 个字的时间分别是：

$t2 = m\,T = 8 \times 100ns = 8 \times 10^{-7}$ (s)

$t1 = T + (m - 1)\tau = 100 + 7 \times 10 = 1.7 \times 10^{-7}$(s)

顺序存储器带宽是：

$W2 = q / t2 = 512 \div (8 \times 10^{-7}) = 64 \times 10^{7}$（位/ S）

交叉存储器带宽是：

$W1 = q / t1 = 512 \div (1.7 \times 10^{-7}) = 301 \times 10^{7}$ （位/ S）

2. 用 64K×1 位的 DRAM 芯片构成 256K×8 位的存储器。要求：

（1） 计算所需芯片数。

（2） 若采用异步刷新方式，每单元刷新间隔不超过 2ms，则产生刷新信号的间隔是多少时间？

参考答案：

（1）256KB / 64K×1 位 = 4×8 = 32 片。

（2）因为每个单元的刷新间隔为 2ms，所以，采用异步刷新时，在 2ms 内每行必须被刷新一次，且仅被刷新一次。因为 DRAM 芯片存储阵列为 64K=256×256，所以一共有 256 行。因此，存储器控制器必须每隔 2ms/256=7.8μs 产生一次刷新信号。

二、高速缓存

1. 直接映射（directed mapped）：

为了将内存数据块放置到高速缓存中，可以将内存地址分几部分看待:

Tag bits	Index bits	Offset bits

Offset bits 是块内偏移，Index bits 用于标记该内存数据块在高速缓存中的组号(set number)，剩下的 tag bits 是该数据块的标记。

如果高速缓存采用直接映射（direct mapped）采用写直达（write through）的更新策略，那么高速缓存中每一行包含的内容为：cache data block, tag, valid bit， 不需要包含 dirty bit.

根据已知信息，请填写下表中的空格部分：

Address size(bits) 内存地址的长度，寻址到 byte	Cache size	Block Size 数据块大小	tag bits tag 位数	Index bits Index 位数	Offset bits 块内偏移位数	Bits per row 每一行的总位数
16	4KB	4B	4	10	2	32+4+1
32	32KB	16B				
32			16	12		
64	2048KB			14		1068

答案：

Address size (bits)	Cache size	Block size	Tag bits	Index bits	Offset bits	Bits per row
16	4KiB	4B	4	10	2	32+4+1
32	32KiB	16B	17	11	4	128+17+1
32	64KiB	16B	16	12	4	128+16+1
64	2048KiB	128B	43	14	7	1068

2、组相联映射(set associative)：

假设某计算机的主存地址空间大小为 64MB，采用字节编址方式。其 cache 数据区容量为 4KB，采用 4 路组相联映射方式、LRU 替换和回写（write back）策略，块大小为 64B。请问：

（1） 主存地址字段如何划分？要求说明每个字段的含义、位数和在主存地址中的位置。

（2）该 cache 的总容量（不仅包括数据区容量）有多少位？

参考答案：

（1）cache 的划分为：4KB = 2^{12}B = 2^4 组×2^2 行/组×2^6 字节/行，所以，cache 组号（组索引）占 4 位。 主存地址划分为三个字段：高 16 位为标志字段（1 分）、中间 4 位为组号（1 分）、最低 6 位为块内地址

即主存空间划分为：64MB = 2^{26}B = 2^{16} 组群×2^4 块/组群×2^6 字节/块

（2）cache 共有 64 行，每行中有 16 位标志、1 位有效位、1 位修改(dirty)位、2 位 LRU 位，以及数据 64B。故总容量为 64×(16+1+1+2+64×8)=34048 位。

3、代码分析与高速缓存的性能：

一个二路组相联映射的高速缓存（ 2-way associative cache ）容量为 128 bytes ， 每个高速缓存块大小为 32 字节（32 bytes per block）. long long 型数据的长度为 8 个字节（8 bytes）. 假定 table 数组的起始地址为 0x0. **以下代码的高速缓存失效率(miss rate)为多少？**

```
int i;
int j;
long long table[4][8];
for (j = 0; j < 8; j++)
    for (i = 0; i < 4; i++)
      { table[i][j] = i + j; }
```

Solution: long long 型数据的长度为 8 个字节（8 bytes）

m m m m m m m m
m m m m m m m m
m m m m m m m m
m m m m m m m m Miss rate = 1

4、平均存储器访问时间（Average Memory Access Time：AMAT）

AMAT 是内存访问的平均（expected）时间，可以用以下公式来估算：
AMAT = hit_time + miss_rate × miss_penalty

- Hit_time: cache hit 时，访问 cache 所花的时间
- Miss_rate: 高速缓存的失效率
- miss penalty：当发生 cache miss 时，需要花的额外的访存时间，所以一次 cache miss 需要花费（hit_time + miss_penalty）的时间

假设高速缓存系统的属性如下，求 AMAT 是多少？

a) L1$ hits in 1 cycle (local miss rate 25%)
b) L2$ hits in 10 cycles (local miss rate 40%)
c) L3$ hits in 50 cycles (global miss rate 6%)
d) Main memory hits in 100 cycles (always hits)

Global miss rate 和 Local miss rate 的定义请参考如下描述：
Global miss rate：
– the fraction of references that miss some level of a multilevel cache
– misses in this cache divided by the total number of memory accesses generated by the CPU

Local miss rate – the fraction of references to one level of a cache that miss
Local Miss rate L2$ = L2$ Misses / L1$ Misses

The AMAT is 1 + 0.25*(10 + 0.4*(50)) + 0.06*100 = 14.5 cycles.
Alternatively, we can calculate the global hit rates for each hierarchy:

- **L1$: 0.75**
- **L2$: 0.25*0.6 = 0.15**
- **L3$: 0.94 – (0.75+0.15) = 0.04**
- **Main Memory: 1 – 0.75 – 0.15 – 0.04 = 0.06**

And the following hit times:

- **L1$: 1 cycle**
- **L2$: 1+10 = 11 cycles**
- **L3$: 1 + 10 + 50 = 61 cycles**
- **Main Memory: 1 + 10 + 50 + 100 = 161 cycles**

Then, AMAT = 0.75*1 + 0.15*11 + 0.04*61 + 0.06 * 161 = 14.5 cycles.

5、虚拟存储器（Virtual Memory）

程序中使用的内存地址是虚拟地址，一个虚拟地址（VA）可以看作两部分：虚页号、页内偏移（page offset），如下图中标示：

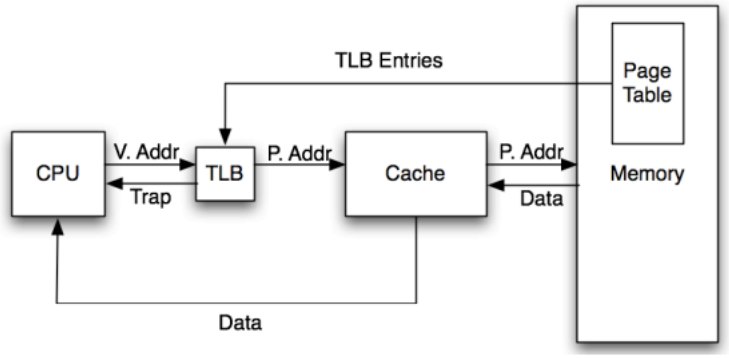
Virtual Page Number	Page Offset

但事实上无论是数据还是指令都是存放在物理内存上的。一个物理地址（PA）也可以看作两部分：物理页号、页内偏移。如下图中标示：

Physical Page Number	Page Offset

如果页大小是 4KB=4096 bytes，那么 page offset 就是 12 位。

将 VA 转换为 PA 会使用到快表（TLB）和页表（Page Table）. 下图示意了 TLB 和页表（p 在内存访问时所处的位置。



每一个进程都有一个页表，页表存储在内存中，操作系统通过设置一个专用寄存器的值，告诉硬件页表在内存中的起始地址。每当切换进程时，操作系统会将要执行的进程的页表起始地址转载到这个专用寄存器中。

页表的结构一般如下：

The Page Table

Index = Virtual Page Number (VPN) (not stored)	Page Valid	Page Dirty	Permission Bits (read, write, ...)	Physical Page Number (PPN)
0				
1				
2				
...				
(Max virtual page number)				

每一个页表项（ page table entry）除了记录虚页号（VPN）和物理页号（PPN）的映射关系之外，还设置了有效位、脏位和权限位：

- “page valid” 有效位：用于标记该虚页是否在内存中；
- “page dirty” 脏位：操作系统需要知道，是否将该内存页更新到磁盘上；
- “permission bits”权限位：用于限制对该页进行某种操作。

快表（TLB）是页表的缓存（cache），假设 TLB 如果采用全相联映射（fully associative）机

制，它的结构如下：

TLB Entry Valid	Tag = Virtual Page Number	Page Table Entry		
		Page Dirty	Permission Bits	Physical Page Number
…	…	…	…	…

回答问题：

1） 如果页表地址寄存器中装入了新的值，TLB 会发生什么操作?

The valid bits of the TLB should all be set to 0. The page table entries in the TLB corresponded to the old page table, so none of them are valid once the page table address register points to a different page table.

2） 某个处理器内存地址长度为 16 位，页大小为 256 byte，TLB 采用全相联映射，总共有 8 个 TLB 表项， 并采用 LRU 替换机制（LRU 位有 3 位，可以表示 8 个 TLB 表项的访问次序。如果 LRU 位的值为 0，表示该页最近刚刚被访问）。
假设当前进程初始时 TLB 的内容如下，并假设该进程访问的所有页既可以读也可以写。

Initial TLB

VPN	PPN	Valid	Dirty	LRU
0x01	0x11	1	1	0
0x00	0x00	0	0	7
0x10	0x13	1	1	1
0x20	0x12	1	0	5
0x00	0x00	0	0	7
0x11	0x14	1	0	4
0xac	0x15	1	1	2
0xff	0x16	1	0	3

假设现在空闲的物理页是：0x17, 0x18, 0x19；
如果接下来，用标记出的访问模式（读或者写）对以下内存地址进行访问：

Access pattern:

Read	0x11f0
Write	0x1301
Write	0x20ae
Write	0x2332
Read	0x20ff
Write	0x3415

请画出完成以上访问后，TLB 的 final state。

VPN	PPN	Valid	Dirty	LRU

Read 0x11f0: hit, LRUs: 1,7,2,5,7,0,3,4
Write 0x1301: miss, map VPN 0x13 to PPN 0x17, valid and dirty, LRUs: 2,0,3,6,7,1,4,5
Write 0x20ae: hit, dirty, LRUs: 3,1,4,0,7,2,5,6
Write 0x2332: miss, map VPN 0x23 to PPN 0x18, valid and dirty, LRUs: 4,2,5,1,0,3,6,7
Read 0x20ff: hit, LRUs: 4,2,5,0,1,3,6,7
Write 0x3415: miss and replace last entry, map VPN 0x34 to 0x19, dirty, LRUs, 5,3,6,1,2,4,7,0

Final TLB

VPN	PPN	Valid	Dirty	LRU
0x01	0x11	1	1	5
0x13	0x17	1	1	3
0x10	0x13	1	1	6
0x20	0x12	1	1	1
0x23	0x18	1	1	2
0x11	0x14	1	0	4
0xac	0x15	1	1	7
0x34	0x19	1	1	0