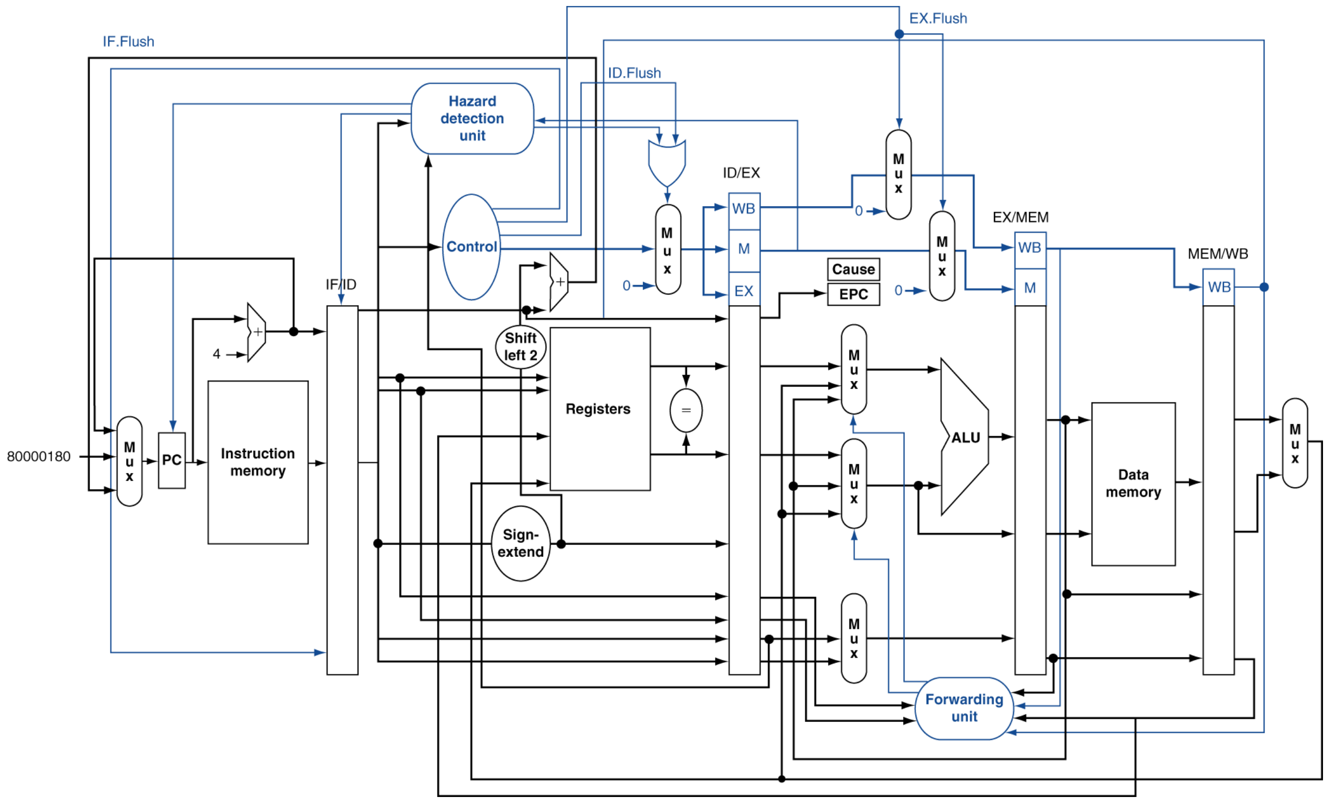
NATIONAL TSING HUA UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE
CS 4100: Computer Architecture
Spring 2018, Final Examination

1. (16%) Consider the pipelined design of the MIPS processor and the code segment shown below.

```
sub     $2,$1,$3
lw      $4,50($2)
add     $4,$6,$2
beq     $4,$2,-5
sw      $4,100($2)
```



(1) (8%) Suppose at any cycle the register file can perform either read (read from two registers) or write (write into a register), but not both.

(a) What type of hazards will this design produce? Use the above code segment to explain when this type of hazards will occur.

(b) To resolve this type of hazards, we let write-to-register take a higher priority than read-from-register. Explain how you will handle the IF/ID and ID/EX pipeline registers, respectively, when the hazard occurs, e.g., setting the control signals.

Ans：

a. structure hazard (between “sub” and ”beq” when “sub” in “WB”)
   control hazard (between “beq” and ”sd” when “beq” in “ID”)

b. solution：兩者都是在 IF/ID ”stalling”
   發生Hazard的話就需要塞入stall，也就是將stage之間的buffer清空，達到暫停inst.的效果。
   ⇨ 讓beq晚點做，所以包含beq在內，之前的inst.都必須停下
   ⇨ Set IF.Flush = 1, let IF/ID flush；Set ID.Flush = 1, let ID/EX flush

(2) (5%) For the following questions, assume that the register file can support two reads and one write at the same cycle. What is the size of the ID/EX pipeline register, excluding the control signals?

Ans：
PC – 64bits, data(rs1) & data(rs2) – 32bits, imm(after sign-extention) – 64bits,
rs1 & rs2 – 5bits, rd – 5bits

(3) (5%) When **`sub`** is at the WB stage and **`add`** at the EX stage, what are the values of the two outputs of the “Forwarding unit”? Give your reasons. (Note: the possible values at each of the two outputs are 0, 1, or 2.)

Ans：
ForwardA(upper) = 01_2 , ForwardB(lower) = 00_2

(4) (11%)
(a) When **`lw`** is at the MEM stage, **`add`** at the EX stage, and **`beq`** at the ID stage, what is the output value of “Sign-extend” in hexadecimal?
(b) Explain why the data hazard between **`lw`** and **`beq`** or between **`add`** and **`beq`** CANNOT be resolved by forwarding?
(c) How can this data hazard be resolved? Explain your answers.

Ans：
a. $-5_2 = 1011_2$ → (sign-extend) → 1111 … 1011 = $\text{FFFF FFFF FFFF FFFB}_{16}$
b. “ld” 要取得data值是在WB，如果沒有stalling只有forwarding ”beq” 此時已在EX，然而 “beq” 在ID時就應該要拿到x4的data，因此只有forwarding沒法解決這個data hazard。
c. When “beq” in ID stage, give IF/ID a stalling to solve this problem.

(5) (5%) It is possible to forward the ALU result from the MEM stage to the ID stage, if **`beq`** is at the ID stage and it is data-dependent on the instruction at the MEM stage. Draw a diagram to show how the outputs of the register file should be modified to take the data forwarded from the MEM stage. (You need not show the forwarding unit.)

Ans：Draw

(6) (5%) We want to extend the IF stage to include a branch target buffer (BTB) with branch predictors. Draw a diagram to show how the input to and output from the PC register should be modified to work with the BTB. Explain how your design work so that the next cycle can fetch instructions from the predicted path after a **`beq`**.

Ans：Draw

(7) (5%) Explain what happen when **`add`** causes an overflow exception at the EX stage.

Ans：
i. Save PC of offending instruction (SEPC)
ii. Save indication of the problem (SCAUSE)

2. Suppose we decide to design a two-way set-associative data cache for the above pipeline. The pipeline uses 32-bit byte addresses and 32-bit words. The data cache consists of 512 sets. Each cache block contains 2 words.

(1) (5%) What is the total size of the cache in bits, including the valid bits, tag bits, and data bits?

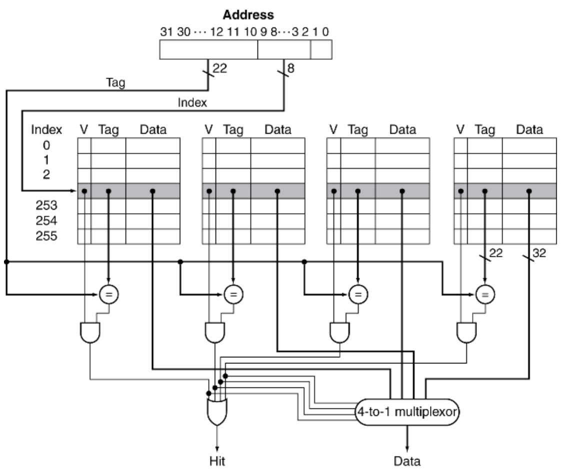
Ans：
index – 9 bits (since $2^9 = 512$), 2 words = 8 bytes = 64 bits → offset – 3 bits,
tag = 32-9-3 = 20 bits, valid – 1 bit, data – 32*2 bits
➔ Total bits = (1+20+32*2)*2*512

(2) (6%) Explain how this cache organization exploits spatial locality of reference, and how it reduces compulsory misses.

Ans：
i. Using it to move blocks consisting of contiguous words to the upper levels.
ii. Using larger block size to reduce compulsory misses.

(3) (8%)
(a) For this cache organization, the time required for a read hit and a write hit are different. Explain why.
(b) On the other hand, a direct-mapped cache with one work per block has almost the same read hit and write hit time. Give you explanations. (Hint: Do we need to check the old data in the cache on a write hit for this cache?)

Ans：
a.



參考左邊這張圖(L5-2 p23)，可以看出read可以同時做read data和compare tag兩件事。也就是說，當compare tag後得出哪個block才是目標，就可以馬上拿到data
⇨ read data和compare tag是平行的
但write的話，則需要先compare tag確認哪個block是目標後，才可以接著寫入data
⇨ write data接在compare tag之後

b. 因為是direct-mapped的關係，所以index和block之間的關係是一對一，可以省略compare tag這項步驟，如此read hit和write hit就分別只剩下read data和write data的動作 => 所需時間相近

(4) (5%) Ideally, the time to perform a cache access should be affected by that access only and not by other accesses. Which write policy, write-through or write-back, is close to the ideal? Give your explanations.

Ans：write-back，因為write-back只需要write在cache的data，不用像write-through還需要access memory去write在memory的data。

(5) (12%) Suppose we want to perform array addition of an array A[2048] (2048 words) with each of the four arrays: B0[2048], B1[2048], B2[2048], and B3[2048]. Results are

accumulated in A[2048]. Assume the program does not access other variables, and arrays A, B0, B1, B2, and B3 are allocated sequentially in the memory by the compiler.
(a) How many misses will occur if we perform A+B0, A+B1, A+B2, and A+B3 in sequence?
(b) Among the misses, how many are conflict or capacity misses?
(c) How will you rewrite the program to eliminate conflict and capacity misses?

Ans：

a. Cache滿之前，每個block都是第一次load進來
=> 前1024個block都是compulsory miss => 512 block是A，512 block是B0
=> 此時，進行到A[512*2] + B0[512*2] = A[1024] + B0[1024]

Cache最多只能放入1024個block => 之後每進來一個block，就會換掉一個block
=> 因為每個block包含2 words，可以提供該array 2次運算
A+B0還剩下1024次運算
=> (1024 / 2) blocks/per array * 2 array = 1024 misses
A+B1、A+B2、A+B3各有2048次運算
=> (2048/2) blocks/per array * 2 array * 3 = 6144 misses

共有1024 + 1024 + 6144 = 8192 misses

b. 所有misses - compulsory miss = conflict or capacity misses = 8192 – 1024 = 7168

c. 先算出B0+B1+B2+B3，再write到A。如此可以避免重複read A的block。
(前作法每次相加，都需要重新read A)

(6) (6%) Suppose the `lw` instruction in Question 1 produces an address of 0x004B50E2 at the EX stage. When it enters the MEM stage, the address is used to fetch a word from the memory. Now, assume that the system runs a virtual memory with a page size of 4096 bytes.
(a) What is the virtual page number of this address?
(b) Explain how the address is translated into the physical address with the help of TLB and page table.

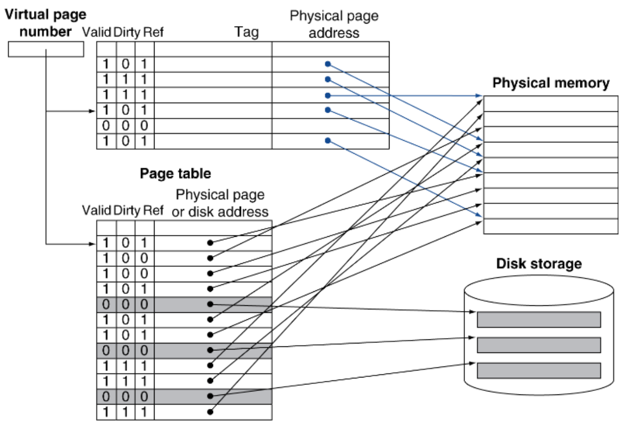
Ans：

Address – 0000 0000 0100 1011 0101 0000 1110 0010

a. 4096 bytes = 2^{12} bytes → use address[11:0] to be as page offset
→ virtual page number is address[31:12] = 0x004B5

b.

TLB : (in Cache, SRAM)



參照左圖
1. virtual address可以拆成virtual page number和page offset
2. virtual page number當作tag，尋找有同樣tag的entry。**若是沒有，則到Page Table**
3. 找到符合的TLB entry後，即可得到physical page address
4. physical page address和page offset合併成physical address
5. physical address即可從physical memory得到block

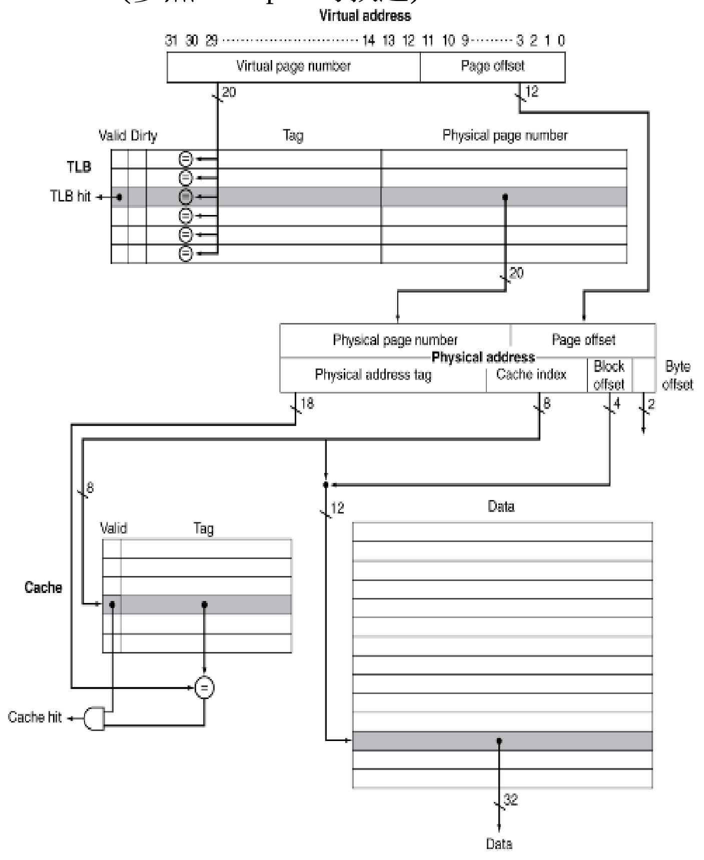
Page Table : (in Physical Memory, DRAM)
1. virtual address可以拆成virtual page number和page offset
2. virtual page number當作index，可以找出對應的Page Table Entry(PTE)，如此就能

得到physical address -> 第一次memory access
3.根據physical address即可從physical memory得到block -> 第二次memory access

(7) (5%) From Question 2(6) above, will the portion of the address that we use to index a set in the data cache be changed before and after virtual address translation? Can virtual address translation in TLB and data access in cache be performed at the same time, instead of sequentially?

Ans：
the portion of the address before virtual address translation : virtual page number
the portion of the address after virtual address translation : physical page number
通常，virtual memory會大於physical memory，所以virtual page number的長度會大於physical page number。

可以，只要使用virtual page number當作TLB和Cache的Tag，就可以同時進行TLB和Cache。(參照L5-3 p23的敘述)


Virtual address
31 30 29 ··· 14 13 12 11 10 9 ··· 3 2 1 0
Virtual page number
Page offset
20
12
Valid Dirty
Tag
Physical page number
TLB
TLB hit
20
Physical page number
Page offset
Physical address
Physical address tag
Cache index
Block offset
Byte offset
18
8
4
2
8
12
Valid
Tag
Data
Cache
Cache hit
32
Data