CMPE 200 Spring 2018
Quiz

Total Score: /20

Instructor: Sheng-Liang Song

Computer Engineering Department, San Jose State University

Quiz	Scores
1. __sync_lock_test_and_set	/2
2. Clock divide by 3	/2
3. Atomic APIs	/2
4. D/Q in Pipeline Register	/2
5. Pipeline Stall and Forwarding	/2
6. Extend the datapath to also support JALR	/2
7. 2-way set-associative cache	/2
8. Variable Size Page Table vs Hashed Page Table	/2
9.Virtual Address, TLB, Cache Tag (Physical Tag, Virtual Tag)	/2
10. Meltdown Paper	/2
Total	/20

Question 10: Read Meltdown Paper & Checkout and Run Examples.

https://github.com/Shengliang/meltdown

https://drive.google.com/drive/u/1/folders/1IccGVN9Oj8fiy0wkUpSG3tzy4El7RWBM

Question 9: Virtual Address, TLB, Cache Tag (Physical Tag, Virtual Tag)

The following worksheet shows:

L1: is virtually tag cache

L2: is physical tag cache

TLB: direct-mapped, and fully associative

L1: direct-mapped and 2-way

L2: direct-mapped and 8-way (?)

https://docs.google.com/spreadsheets/d/1hNXt-yOGN4l-nKOcEPXmLYcl9syXZ0XV7iO7XcJc2Uc/edit#gid=0

https://www.youtube.com/watch?v=YbE91A7dNAs

Question 8: Variable Size Page Table vs Hashed Page Table

Given:

1) **Page Size is 64KB**
2) **32 Virtual Address Bus**
3) **Physical Memory 64GB**

What is 1-level linear page table size per user space application?

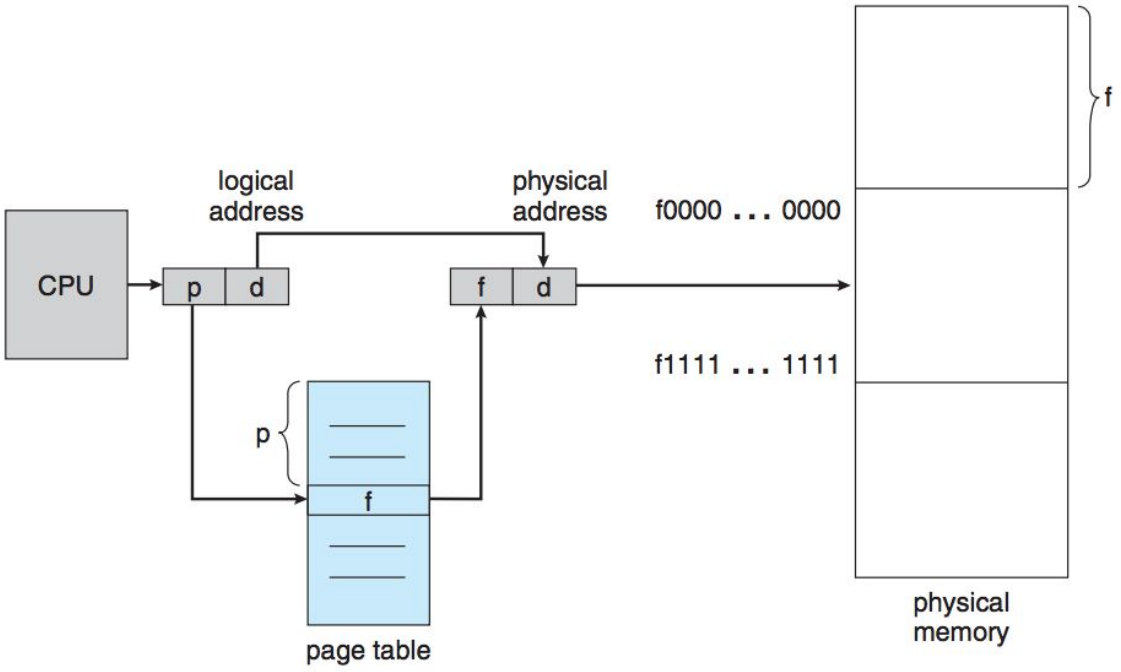


Figure 8.10 Paging hardware.

Page Size 64KB => d = 16 bit page offset

Virtual Page Number => p = 32 bit - 16 bit = 16 bits.

Page Table Entry (PTE): 4B

- A bit to indicate if a page exists (1 bit)
- PPN, physical page number for a memory-resident page (20 bits)
- Status bits for protection and usage (N bits)

64GB => 36 bit physical address

Physical Memory frame number (Physical Page Number) => f = 36 bit - 16 bit = **20 bits**

If a PTE is 32 bits (4 Byte), then

1-level linear page table size = 2^16 * 4B = 64K * 4B = 256 KB

// That can address 2^16 * 64KB = 64K*64KB= 2^12MB = 4GB memory

// or 32-bit virtual address can address 4GB memory.

What is 1-level linear page table size per user space application if the virtual address is 64-bit?

Virtual Page Number => p = 64 bit - 16 bit = 48 bits

1-level linear page table size = 2^48 * Page Table Entry Size = 2^48 * 4B = 2^50 B = 1PB. // That is not practical.

What is hashed page table size per user space application if the virtual address is 64-bit, and a hash function map 48-bits VPN to 16-bits hash-code?

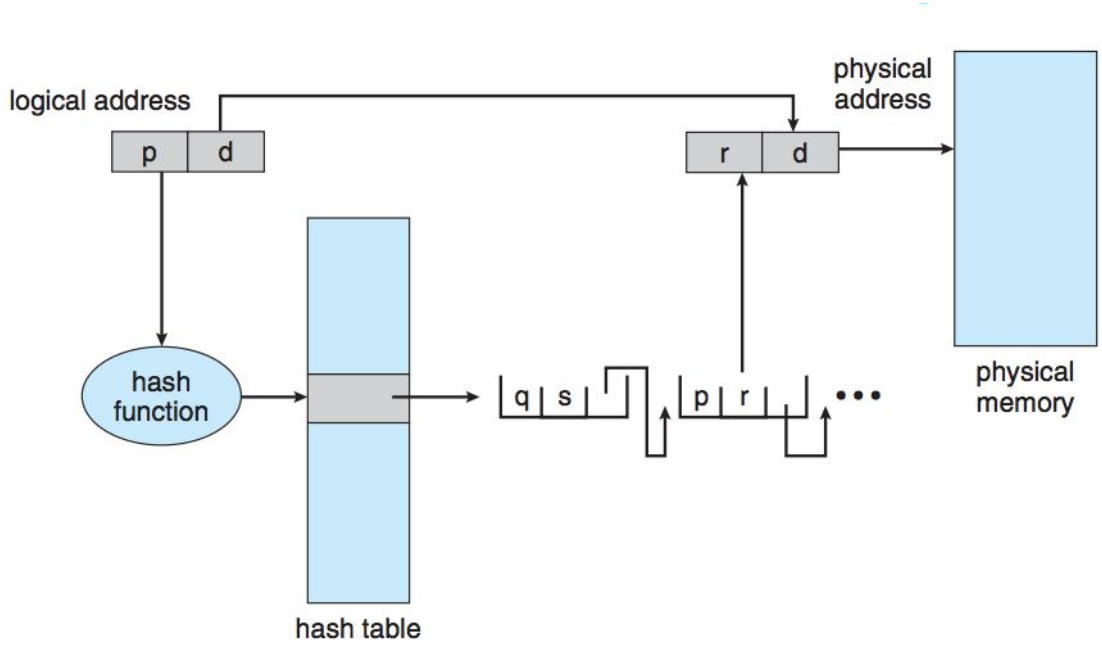


Figure 8.19 Hashed page table.

```
uint16_t get_hash_code(uint64_t vpn) {
        // Input: vpn = 0xFA44332211CE;
        // Output: hc   = 0xFACE;
        return ((vpn >> 32) & 0xFF00) | (vpn & 0xFF);
}

uint64_t get_vpn_from_hash_code_and_tag(uint16_t hash_code, uint32_t tag) {
        // Input: hc   = 0xFACE;
        // Input: tag  =  0x44332211;
        // Output: vpn = 0xFA44332211CE;
        return  ((hash_code & 0xFF00) << 32) | (tag << 8) | (hash_code & 0xFF));
}
```

Hashed Page Table Entry (PTE): = About 8 Bytes

- A bit to indicate if a page exists (1 bit)
- // VPN, virtual page number (48 bits)

- Virtual page number tag: **32 bits**
- Physical Memory frame number (Physical Page Number): f = 36 bit - 16 bit = 20 bits
- Status bits for protection and usage (N bits)
- Next pointer (X bits) // Assume no next pointer.

Assume 1 hash entry saves 4 PTE. 8B * 4=32B (without next pointer)

[PTE] [PTE] [PTE] [PTE]

Hashed **page table size = 2^16 * 32B = 2^21B = 2 MB.**

Figure 8.24 illustrates a x86 PAE system with 4-KB pages.

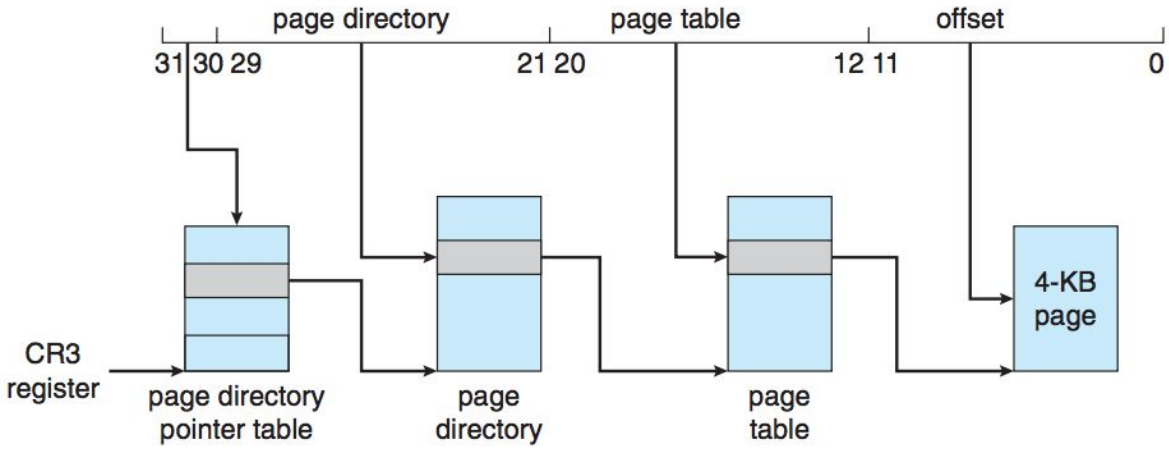


Figure 8.24 Page address extensions.

```
struct address_t {
   uint32_t page_directory_pointer_table_offset : 2;
   uint32_t page_directory_offset : 9;
   uint32_t page_table_offset : 9;
   uint32_t page_offset : 12;
};
```

Page Size: 4KB (12 bit page offset)
64GB => 36 bit physical address .
Physical Page Number size = 36 bit - 12 bit = 24 bits

1) Size of page directory pointer table: 4 * 4B = 16B
2) Size of page directory: 512 entry ; 512 * 8B = 4KB; Total Size: 4KB *4 = 16KB
3) Size of page table segment: 512 entry; 512 *8B = 4KB;
4) Size of page table: 4KB * 512 = 2MB
5) Addressable memory from CR3: 4KB * 512 * 512 * 4 = 4GB

Physical Address Extensions (PAE)

http://www.superspeed.com/common/RdRp-Mem.pdf

https://en.wikipedia.org/wiki/Physical_Address_Extension

http://slideplayer.com/slide/7316543/

http://iips.icci.edu.iq/images/exam/Abraham-Silberschatz-Operating-System-Concepts---9th2012.12.pdf

Question 7: Cache Questions

https://docs.google.com/document/d/1THTZNaD5SliZa2AF4J-Y3MCD7bvJQnPCtA0tWFrIqPY/edit

Question 6

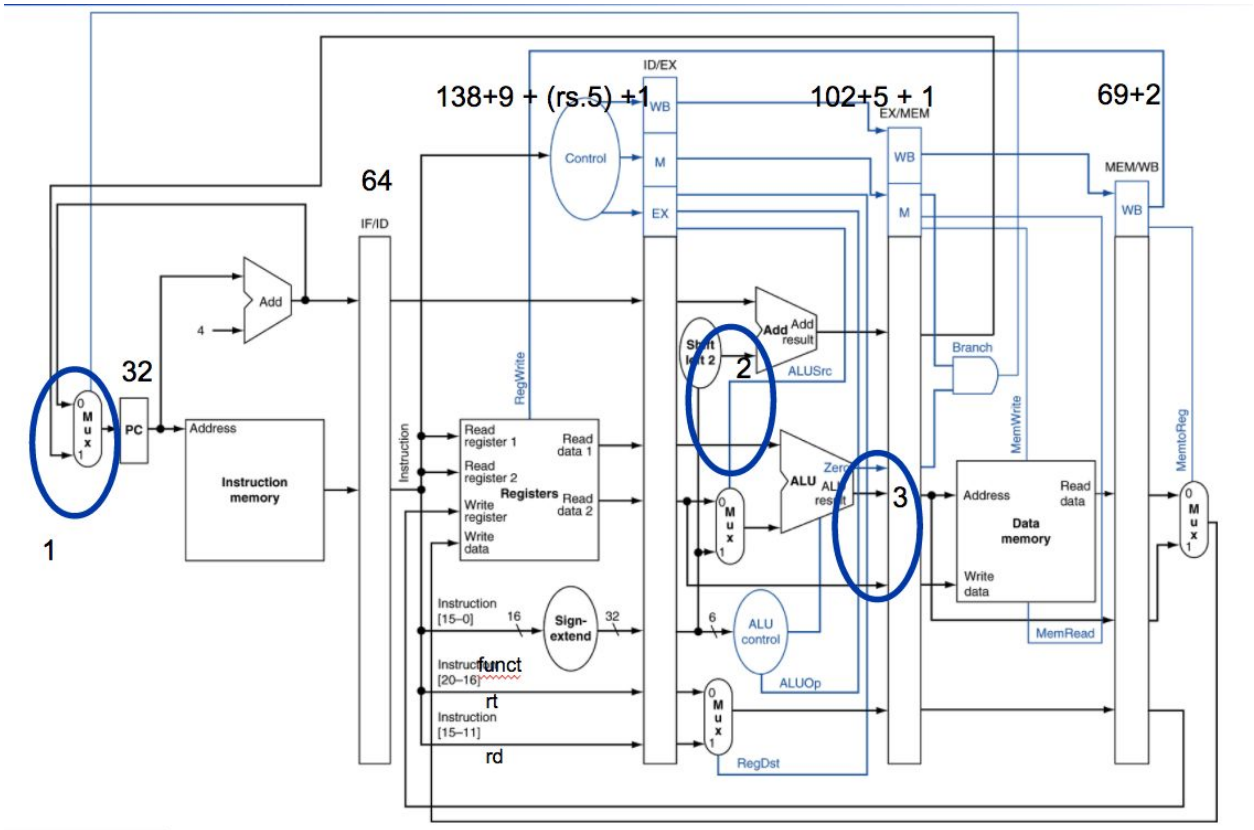
Consider the following datapath that can execute a limited set of instructions (LW, SW, R-type instructions, BEQ, and J). Please extend the datapath to also support JALR (Jump with and link register) instruction. RD = PC + 8; PC = RS. Please add/modify logic in (1), (2), and (3).

Jump and link register

`jalr rs, rd`

0	rs	0	rd	0	9
6	5	5	5	5	6

Unconditionally jump to the instruction whose address is in register `rs`. Save the address of the next instruction in register `rd` (which defaults to 31).



Question 5

a) Let's run **your wrapper function** code on a five-stage pipeline architecture. We use a baseline register file where a newly written register value can be read in the subsequent cycle. Show the execution of above code by filling the timing diagram in the next page.

Assuming no forwarding logic.

b) Repeat a. Now try to add hardware forwarding logic to eliminate stalls. (10 pts)
We still use a baseline register file where a newly written register value can be read in the subsequent cycle.

Please include the source and the destination of your forwarding logic.

With and without **branch delay slot** examples

1) Assume branch outcome is determined at EXE stage (in the book; assume **without branch delay slot**)

// if branch is taken; next PC = PC + 4 + offset = 104 + 4 + 200 = 308

100: lw $7, 0($8)	F	D	E	M	W													
104:beqz $7,+200		F	D	*	E	M	W											
108:addu $6,$7,$0			F	*	D	~~E~~	~~M~~	~~W~~				**Killed when branch is taken**						
10C:addiu $9,$8,4					F	~~D~~	~~E~~	~~M~~	~~W~~			**Killed when branch is taken**						
308:subiu $9,$8,4						F	D	E	M	W								

2) Assume branch outcome is determined at ID stage (if using gcc compiler; assume with **branch delay slot)**

// if branch is taken; next PC = PC + 4 + offset = 104 + 4 + 200 = 308 (assume 1 **branch delay slot)**

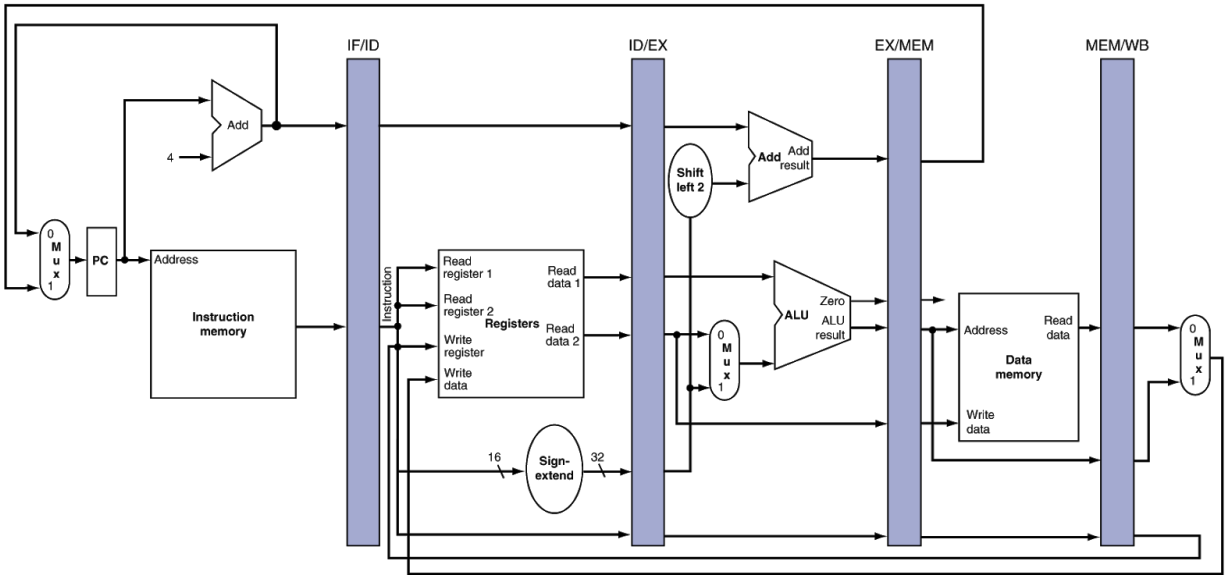
100:lw $7, 0($8)				F	D	E	M	W										
104:beqz $7, +200					F	*	*	D	E	M	W							
108:addu $6,$7,$0								F	D	E	M	W		//**branch delay slot:** always execute				
308: subiu $9,$8,4									F	D	E	M	W					

// if branch is not taken, next pc = PC + 8 = 104 + 8 = 0x10C (assume with 1 **branch delay slot)**

100:lw $7, 0($8)				F	D	E	M	W										
104:beqz $7, +200					F	*	*	D	E	M	W							
108:addu $6,$7,$0								F	D	E	M	W		//**branch delay slot:** always execute				
10C:addiu $9,$8,4									F	D	E	M	W					

Q4: The following diagram is a single-cycle pipeline diagram that we discussed.

add $14, $5, $6	lw $13, 24 ($1)	add $12, $3, $4	sub $11, $2, $3	lw $10, 20($1)
Instruction fetch	Instruction decode	Execution	Memory	Write-back



a.) Please please run your Q3 wrapper function code through the pipeline and decode all values (D and Q) in EXE/MEM pipeline register in the above diagram.

Pipeline Register	**EXE/MEM.D**	**R**	**EXE/MEM.Q**
Instruction Machine Code			
ISA Format			
opcode=?			
Next PC[31:0]			
ALU Result[31:0]			
RT Value[31:0]			
Dest Reg[4:0]			

CH2: Quiz

Please reference list_tail_append() **in Question 1 and generate MIPS32 assembly and machine code for the following APIs. Your output should be similar to** list_tail_append() output.

You may use gcc tools: mips-mti-elf-gcc and mips-mti-elf-objdump.

Reference: https://github.com/Shengliang/language/blob/master/c/atomic_api.c

Pick # from the following list:

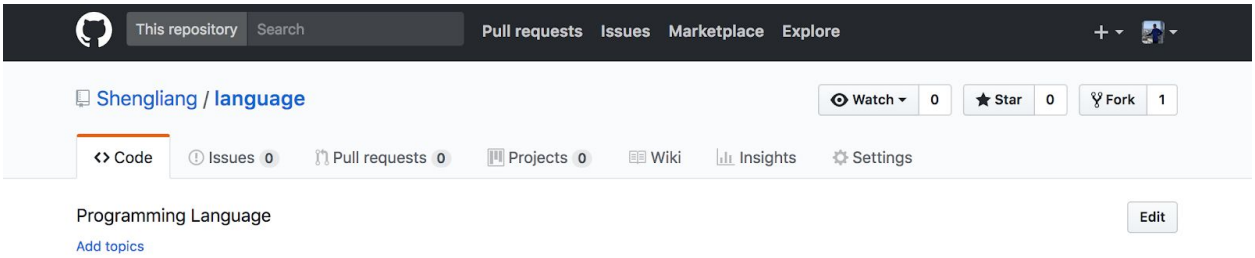
N = Your ClassIndex % 14 + 1
Where N = {1,2,3,4,5,6,7,8,9,10,11,12,13,14}

https://git.linaro.org/kernel/linux-linaro-stable.git

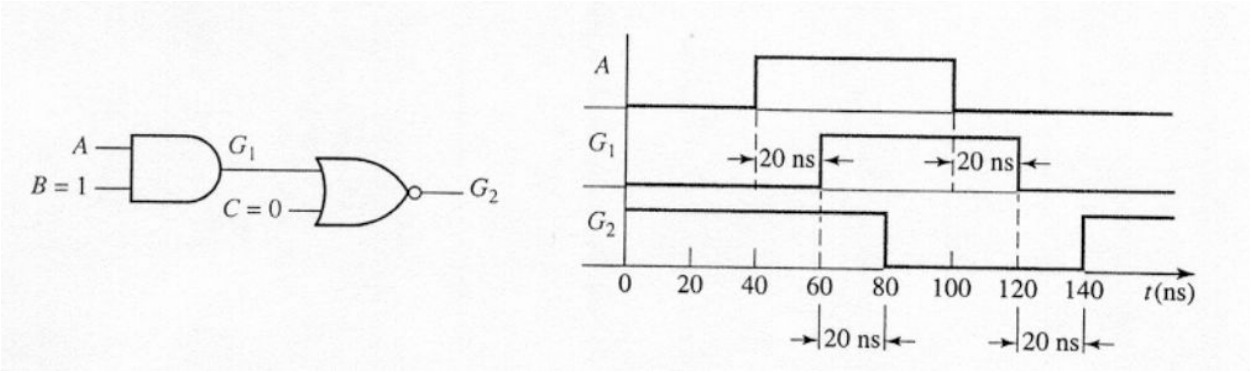
1. type __sync_fetch_and_add (type *ptr, type value, ...)
2. type __sync_fetch_and_sub (type *ptr, type value, ...)
3. type __sync_fetch_and_or (type *ptr, type value, ...)
4. type __sync_fetch_and_and (type *ptr, type value, ...)
5. type __sync_fetch_and_xor (type *ptr, type value, ...)
6. type __sync_fetch_and_nand (type *ptr, type value, ...)
7. type __sync_add_and_fetch (type *ptr, type value, ...)
8. type __sync_sub_and_fetch (type *ptr, type value, ...)
9. type __sync_or_and_fetch (type *ptr, type value, ...)
10. type __sync_and_and_fetch (type *ptr, type value, ...)
11. type __sync_xor_and_fetch (type *ptr, type value, ...)
12. type __sync_nand_and_fetch (type *ptr, type value, ...)
13. bool __sync_bool_compare_and_swap (type *ptr, type oldval, type newval, ...)
14. type __sync_val_compare_and_swap (type *ptr, type oldval, type newval, ...)
15. type __sync_lock_test_and_set (type *ptr, type value, ...)
16. void __sync_lock_release (type *ptr, ...)

1) **Create your github account**
2) **Add your google drive link and github link in ClassIndex Sheet**
3) **Go to https://github.com/Shengliang/language**
4) **Click fork**
5) **git clone https://github.com/<your_github_login>/language.git**
   Ex: https://github.com/Shengliang/language.git

6) https://github.com/Shengliang/language/blob/master/c/atomic_api.c

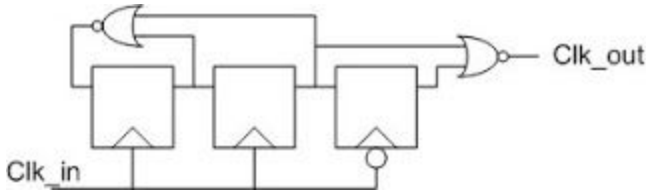


Appendix Review: Question 2 (1 points)



The above example assumes that tPLH and tPHL equal 20 ns for both AND and NOR gate. (50MHz : 20ns)

Reference:
http://www.ece.ucsb.edu/Faculty/Johnson/ECE152A/handouts/L4%20-%20Propagation%20Delay,%20Circuit%20Timing%20&%20Adder%20Design.pdf



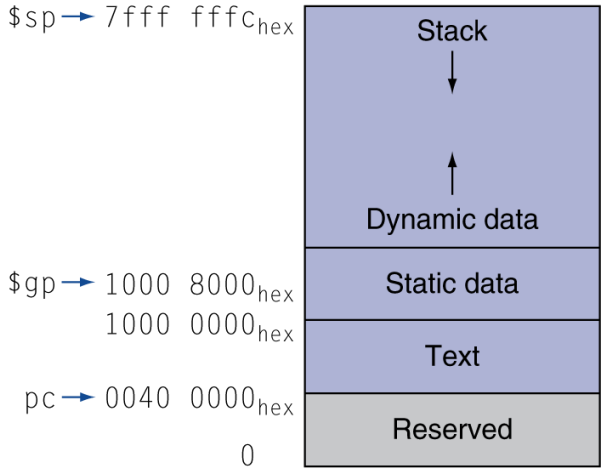
If clk_in is 1GHz, then what is the waveform for clk_out?

Assume “gate” delay: t = (1/50) * Period = (1 ns)/50 = 1000ps/50 = 20ps

Assume D-Flip-Flop delay: T = 40ps

CH2: Question 1 (1 points)

What are the printf output based on the following memory map? Please pick a address in the correct memory regions {Text, Static Data, Dynamic Data, Stack}.




```
#include<stdio.h>
#include<stdlib.h>
struct node_t {
  unsigned int value;
  struct node_t *next;
};
void print(struct node_t* n) {
  printf("print: %p\n", print);
  printf("&n: %p\n", &n);
  printf("&(n->value): %p\n",   &(n->value));
  printf("&(n->next): %p\n",    &(n->next));
  printf(" (n->next): %p\n",     (n->next));
}

int main (void)  {
      struct node_t* n = (struct node_t*) malloc(sizeof(struct node_t));
      n->value = 0xFACE;
      n->next  = n;
      print(n);
      free(n);
      return 0;
}
```

I use the following command to generate objdump file.

mips-mti-elf-gcc -std=c99 -O3 list_append_v1_0.c

mips-mti-elf-objdump -D a.out > a.txt

```
20 void list_tail_append(struct list_t* list, struct node_t* n) {
21
22      // Check and acquire the lock, if available, else keep on checking
23      while (__sync_lock_test_and_set(&list->lock,1));
24
25      if (list->tail ==  NULL && list->head == NULL) {
26            list->head = n;
27            list->tail = n;
28      } else {
29            list->tail->next = n;
30            list->tail = n;
31      }
32
33      // Release the lock after execution of critical section
34      __sync_lock_release(&list->lock);
35 }
```

1) Which assembly line(s) are matching this GNU API ?
__sync_lock_release(&list->lock) (4 point)

2) Which assembly line(s) are matching this line C code? (6 point)
while (__sync_lock_test_and_set(&list->lock,1));

```
0040028c <list_tail_append>:
  40028c:       24820008        addiu   v0,a0,8
  400290:       c0430000        ll      v1,0(v0)
  400294:       24010001        li      at,1
  400298:       e0410000        sc      at,0(v0)
  40029c:       1020fffc        beqz    at,400290 <list_tail_append+0x4>
  4002a0:       00000000        nop
  4002a4:       0000000f        sync
  4002a8:       1460fff9        bnez    v1,400290 <list_tail_append+0x4>
  4002ac:       00000000        nop
  4002b0:       8c820004        lw      v0,4(a0)
  4002b4:       10400007        beqz    v0,4002d4 <list_tail_append+0x48>
  4002b8:       00000000        nop
  4002bc:       ac450004        sw      a1,4(v0)
  4002c0:       ac850004        sw      a1,4(a0)
  4002c4:       0000000f        sync
  4002c8:       ac800008        sw      zero,8(a0)
  4002cc:       03e00008        jr      ra
  4002d0:       00000000        nop
  4002d4:       8c830000        lw      v1,0(a0)
  4002d8:       1460fff8        bnez    v1,4002bc <list_tail_append+0x30>
  4002dc:       00000000        nop
  4002e0:       ac850000        sw      a1,0(a0)
  4002e4:       1000fff7        b       4002c4 <list_tail_append+0x38>
  4002e8:       ac850004        sw      a1,4(a0)
```