CS61C Sample Final

1 Opening a New Branch

```
struct market {
      char *name;
      struct item *inventory; // array of items
      int invSize; // size of array
};
struct item {
      int id;
      int price;
};
typedef struct market Market;
typedef struct item Item;
```

1) Consider the structures above which describe a Market and its inventory. Complete the function copy below, which makes a complete copy of the market structure and all of the data contained within it. None of the fields in the new copy should reference data in the original copy. Feel free to use the following functions:

```
char *strcpy(char *dst, const char *src);
size_t strlen(const char *str);
```

```
Market* copy(Market* orig) {
    Market* result = malloc(sizeof(Market));
    result->invSize = orig->invSize
    result->name = malloc(sizeof(char) * (strlen(orig->name) + 1));

    strcpy(result->name, orig->name);
    result->inventory = malloc(sizeof(Item)*orig->invSize);

    for (int i=0; i<orig->invSize; i++) {
       result->inventory[i].id = orig->inventory[i].id;
       result->inventory[i].price = orig->inventory[i].price;
    }

    return result;
}
```

2 Thread Level Parallelism

For the following snippets of code below, Circle one of the following to indicate what issue, if any, the code will experience. Then provide a short justification. Assume the default number of threads is greater than 1. Assume no thread will complete before another thread starts executing.

Assume arr is an int array with length len.

a)

```
// Set all elements in arr to 0
int i;
#pragma omp parallel for
for (i = 0; i < len; i++)
      arr[i] = 0;
```

Sometimes incorrect Always incorrect Slower than serial **Faster than serial**

Faster than serial – for directive actually automatically makes loop variable private, so this will work properly. Justification needed to mention that the for directive splits up the iterations of the loop into continuous chunks for each thread, so no data dependencies or false sharing.

b)

```
// Set element i of arr to i
#pragma omp parallel
for (int i = 0; i < len; i++)
      arr[i] = i;
```

Sometimes incorrect Always incorrect **Slower than serial** Faster than serial

Slower than serial – there is no for directive, so every thread executes this loop in its entirety . 3 threads running 3 loops at the same time will actually execute in the same time as 1 thread running 1 loop, so credit for justification was only given if there was a mention of parallelization overhead or possible false sharing.

c)

```
// Set arr to be an array of Fibonacci numbers.
arr[0] = 0;
arr[1] = 1;
#pragma omp parallel for
for (int i = 2; i < len; i++)
      arr[i] = arr[i - 1] + arr[i - 2];
```

Sometimes incorrect **Always incorrect** Slower than serial Faster than serial

Always incorrect (if len>4) – Loop has data dependencies, so first calculation of all threads but the first one will depend on data from the previous thread. Because we said “assume no thread will complete before another thread starts executing,” then this code will always be wrong from reading incorrect values.

3 Huge Pages

High performance applications commonly find themselves bogged down by the virtual memory subsystem when using large data sets. One solution is for the operating system to selectively allocate larger (huge) pages to back the virtual memory of such applications. We will look at one possible way these huge pages could be implemented. For this problem, we are using a 32-bit virtual and 32-bit physical address space, and want to support **8MiB huge pages**, with a normal page size of 4KiB.

To start off, we add a bit to each page table entry to indicate if it is a huge page. When this bit is set to 1, it means that the PPN is that of a 8MiB page in physical memory, and that this entry overrides the next 2047 entries for address translation. We require the virtual addresses of huge pages to be 8MiB aligned, so it's only legal for every 2048th page table entry to be marked as huge.

Example page table, where fields marked X are not used in address translation.

PPN	valid	dirty	huge
7342	1	0	0
6422	1	0	X (illegal)
1423	0	0	X (illegal)
...	...	...	...
9241	1	0	1
X (part of hugepage)	X	X	X (illegal)
X (part of hugepage)	X	X	X (illegal)

a) Fill out the number of bits needed for each field in the page table.

PPN: 20 valid: 1 dirty: 1 huge: 1

b) With a fully-associative 16 entry TLB, what is the maximum amount of memory normally addressable with just cached page entries? What does this increase to if we allow huge pages? Write your answer in IEC format.

64KiB, 128MiB

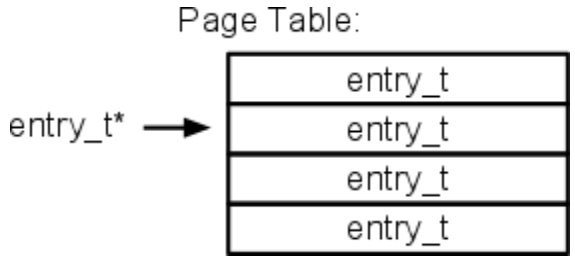
c) List a reason why huge pages might:

speed up translation	waste memory	slow down translation	be better for DMA
less TLB misses - each TLB entry spans a greater amount of memory	internal fragmentation - programs might not use all of the memory in a huge page	there is extra logic added to TLB lookups and page table lookups	can transfer larger amounts of data because huge pages are contiguous in physical memory

d) Describe what hardware modifications are required to allow for huge page support. In particular, what additional checks are necessary during a TLB lookup?
TLB needs extra bit for huge page flag, also need to check if virtual address goes to a huge page or normal sized page during a TLB lookup.

e) Since writing a huge page to disk can take a while, it is common for operating systems to split a huge page into smaller pages when swapping out such a page. Implement split_hugepage(), which is given a pointer to the huge page's entry. You can assume that the page table is an array of entry_t's as defined below, and that the physical huge page can be divided up without moving it. The OS will take care of TLB invalidation after this function is called.

```
// definition of a page table entry
typedef struct entry {
      bool valid;
      bool huge;
      bool dirty;
```

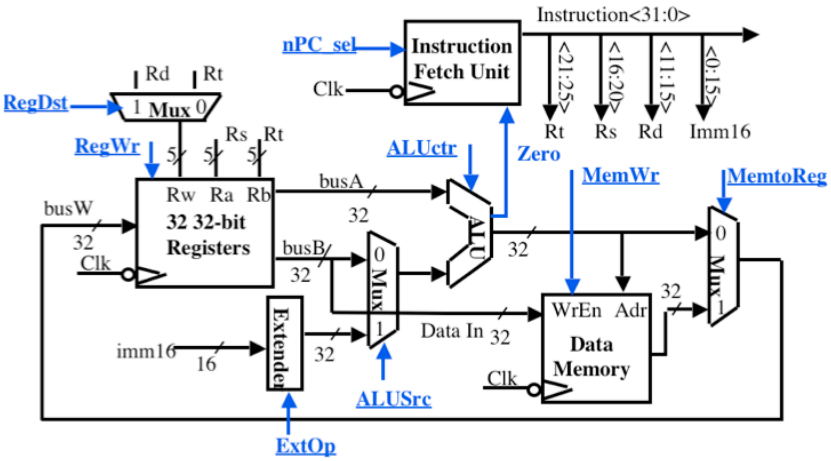


```
        int ppn;
} entry_t;

void split_hugepage(entry_t *huge_ent) {
        huge_ent->huge = 0;
        // your code (HINT: huge_ent + i points to the ith entry after huge_ent)

        for (int i=0; i < 2048; i++) {
                entry_t *ent = huge_ent + i;
                ent->valid = 1;
                ent->huge = 0;
                ent->dirty = huge_ent->dirty;
                ent->ppn = huge_ent->ppn + i;
        }
}
```

4 Triple Add



Above is the **single-cycle** MIPS datapath you all know... ***Ignore pipelining for this question.*** Your job is to modify the diagram to support this new MIPS instruction:

```
add3 rd rs rt
```

What this instruction does is that it adds the three number R[rs], R[rt], R[rd] and stores them back into R[rd].

a) Which MIPS instruction type would be best to represent `add3` and what is the register transfer language for the instruction (don't forget to update the PC)?

b) Change as *little as possible* in the datapath above **(draw your changes right in the figure)** to enable `add3` and list all changes below. Your modification may use muxes, wires, new control signals, and an **additional adder, but nothing else**. Assume you can modify register file to have three outputs: BusA, BusB, and BusC where the value on BusC is specificed by rd. You may not need all boxes.

(i)	Place an adder just outside of the register block and give it inputs BusA and BusC.
(ii)	Mux the above with the value of BusA and give it a control signal called add3.
(iii)	
(iv)	

c) We now want to set all the control lines appropriately. List what each signal should be: an intuitive name (for ALUctr, the operation, for nPC_sel a simple description of the next PC) **or** {0, 1, x – don't care}. Include any new control signals you added.

RegDst	RegWr	nPC_sel	ExtOp	ALUSrc	ALUctr	MemWr	MemtoReg	add3	
1	1	+4	X	0	add	0	0	1	

d) You are trying to sell the value of this instruction. Give a situation in one sentence where this instruction

would be valuable (would allow you to do the task faster than classical mips).

If you are trying to sum a large number of integers.

e) After you make these changes to the datapath, your boss tells you he now has to decrease the frequency of the clock for the CPU to work correctly. What did you do?

You had to extend the period of the clock because the delay of the new adder block might have lengthened the longest pipeline stage.

5 Synchronous Digital Circuits

a) You are an intern at a massive hardware firm. Your first task is to design a “prime number checker.” The circuit you must design with take as input an integer between 0 and 7. It will have three input bits, corresponding to the relevant integer. Your circuit should output 1 if the integer is prime and 0 otherwise. Neither zero nor 1 is prime.

Complete the Truth Table below. A, B and C correspond the 4’s place, 2’s place and 1’s place respectively

Input A	Input B	Input C	Output X
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

b) Write the Boolean expression formed by the output in Sum-of-Products form. Reduce it to its simplest form.
~ab~c + ~abc + a~bc + abc
~ab(~c+c) + ac(~b+b)
~ab+ac

c) Construct the circuit with the fewest gates using only AND, OR and NOT gates

d) Consider a system that can output a value between 0-3 with the ability to increment and decrement. This system will have two 1-bit inputs: increment and decrement and a 2-bit output (the count).

- If increment is high, the count should increase by one for the next cycle (wrap around if necessary).

- If decrement is high, the count should decrease by one for the next cycle (wrap around if necessary)
- If neither is high, the system should stay at the same value
- They will never both be high at the same time

Draw a state machine for this system with appropriate transitions for each input pair.

6 Who invited these people again?

Louis Reasoner writes the following self-modifying code:

```
foo:    la   $t0, modify
        sll  $a0, $a0, 11
        lw   $t1, 0($t0)
        addu $t2, $t1, $a0
```

```
        sw   $t2, 0($t0)
modify: addu $0, $0, $a1
        sw   $t1, 0($t0)
        jr   $ra
```

i) What happens if we call foo with $a0=15 and $a1=15? (3pt)

addu $0 $0 $a1 is changed to addu $t7 $0 $a1

For each of the following questions, CIRCLE either a, b, c, d, or e.
ii) Which set of inputs will permanently change the behavior of the program? (3pt)

a) $a0=3 and $a1=3
b) $a0=5 and $a1=5
c) $a0=7 and $a1=7
d) $a0=9 and $a1=9
e) $a0=11 and $a1=11

This changes the intruction at “modify” to addu $t1,$0,$a1, which will cause the value “0x9” to get written to memory at “modify” instead of the original instruction

iii) Which set of inputs will always cause a bus error? (3pt)

a) $a0=7 and $a1=7
b) $a0=8 and $a1=7
c) $a0=7 and $a1=8
d) $a0=8 and $a1=8

This changes the instruction at modify to addu $t0, $0, 7. Since $t0 is the base address in the next instruction and no longer word aligned, a bus error will result.

iv) Alyssa P. Hacker has figured out how foo works and wants to use the program for an unintended purpose – copying $t3 to $t4. Which set of inputs should she choose? (She could have just executed addu $t4 $0 $t3 to achieve the same effect). (3pt)

a) $a0=0x000000CB and $a1=0x000000CB
b) $a0=0x000000CC and $a1=0x000000CC
c) $a0=0x000000CD and $a1=0x000000CD
d) $a0=0x000000DB and $a1=0x000000DB
e) $a0=0x000000DC and $a1=0x000000D

The question hints that changing the instruction to addu $t4, $0, $t3 will suffice. The object here is to figure out what data value, when shifted by 11, should be added to the intruction “addu $0, $0, $0” to turn it into “addu $t4, $0, $t3.”
Encoding of original instruction:
000000 00000 00101 00000 00000 100001
Encoding of needed instruction:

000000 00000 01011 01100 00000 100001
The difference between these is 0xCC shifted left by 11 - $a0 needs to be set to 0xCC. The value in $a1 doesn’t matter (the register is never read)