cs61c Summer 2011 Final Solutions

Q1 Potpourri (M/F)

Circle whether each of the following statements is True or False. Provide a conclusive argument or counter-example to justify your response. No credit will be given without justification.

2 points per question. A correct, "conclusive" explanation was required to receive full credit.

a) **T / False** in a function that makes lots of function calls, it is more efficient to save local variables in temporary registers than in saved registers

Justification: The function must save and restore temporary registers to and from the stack to preserve their values across function calls. If using saved registers, the function would only need to save and restore the variables once.

There are a few rare cases where using temporary variables might be more efficient (local variables only ever needed before the first function call) – describing one of these cases was also worth credit.

People received no credit for simply stating that temporary registers are volatile.

b) **True / F** Increasing the physical size (radius) of a disk while maintaining the same RPM will increase seek time.

Justification: Seek time is the time it takes to move the disk head from one track to another (inward and outward along the radius of the disk). Increasing disk radius increases the distance the head might have to travel between tracks and thus the seek time.

c) **T / False** Enforcing a Hamming distance of n between codewords means only one-nth of all possible bit strings are valid.

Justification: Best shown with a counterexample. Consider enforcing a hamming distance of 3 with 3 bit strings. Only two strings at a time are valid, for example, 000 and 111, or 010 and 101, etc. 2 valid out of 8 total bit strings != 1/3.

d) **T / False** Let `A = 0xFE7F0000` and `B = 0xFE7F0001. B > A` in unsigned, 2's complement, and floating point.

Justification: True in both unsigned and two's complement. In floating point, the magnitude of B is larger (due to its larger signifcand field), but the numbers are negative, so B < A.

To receive credit, you had to point out specifically that the floating point case was false. Partial credit was given to people who said both floating point and 2's c were false, who who said two's C was true and had a minor error in their discussion of the floating point case.

e) **True / F** The following code will never crash:

```
char *s = malloc(5*sizeof(char));
s = "Hello world";
```

Justification: This code sets s to the address of “Hello world” in static memory. s is not being deferenced; no memory in or around the malloc’d region is being assigned.

This was a tricky question that very few people got. Most people answered that there is not enough room to store “Hello world” where s points, or that the mailoc could fail and cause the program to crash.

f) **T / False** Assembling a program will always increase its instruction count.

Justification: A program containing only TAL instructions (no pseudoinstructions) will retain the same instruction count. People received nearly full credit for answering True and stating how the assembler maps pseudoinstructions to one or more TAL instruction.

To receive partial credit, the student had to show they knew what Assembling was (Assembling is not compiling!)

Q2 Warehouse Scale Questions (F)

a) Name two reasons to replicate data in a Warehouse Scale Computer.

+1 for one correct response, +2.5 for two correct responses.

Accepted responses:

- Fault tolerance (dependability, data restoration, protection against data loss)
- Accessibility (faster response to high demand/"hot spots", geographic locality)
- The ability to use parallelism on same data

b) Name one reason why the MapReduce model of programming requires that the Mapper and Reducer functions do not have side-effects (lasting effects beyond creating output as a function of input).

Worth 2 points, partial credit given in some cases.

- Promotes a simple, easy-to-parallelize model of programming
- Easier to restart a task if it fails because it has no external influence outside of its own processing.

c) Name two ways that multicore systems maintain cache coherence during writes.

+1 for one correct response, +2.5 for two correct responses.

Accepted responses:

- Invalidate on cache write
- Update other caches on write (using interconnect bus)
- Snooping
- Use a single cache

Partial credit:

- Name-dropping MOESI earned you 0.5 points

Incorrect responses:

- Write-back/write-through
- Unified L3 cache does not solve cache coherence among L1 and L2 caches
- Data concurrency methods (test-and-lock) are software-specific and would need to be applied to EVERY BLOCK of data and EVERY PROGRAM on your computer

Q3 Opening a New Branch (M)

```
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) { //8 points
      int i;
      Market *result = malloc(sizeof(Market)); //a. (see notes below)
      result->invSize = orig->invSize //b
      result->name = malloc(sizeof(char)*(strlen(orig->name)+1); //c
      strcpy(result->name,orig->name); //d
      result->inventory = malloc(sizeof(Item)*orig->invSize) //e
      for(i=0;i<orig->invSize;i++) {
            result->inventory[i].id = orig->inventory[i].id; //f
            result->inventory[i].price = orig->inventory[i].price;
      }
      return result; //g
}
```

```
a) .5 points
b) .5 points
c) 1 point for correct malloc statement, 1 point for null terminator
Technically you don't need the sizeof(char), it's part of the C standard
that a char is always 1 byte.
d) 1 point. strcpy DOES copy the null terminator (no penalty was given for
adding it manually)
e) 1 point.
f) 3 points for the for loop. -1 point for every error in the assignment
statements (extra/too few dereferences, etc).
Also accepted was struct assignment (result->inventory[i] = orig
->inventory[i]), which I did not explicitly mention in class.

Special cases:
No malloc statements – Question graded out of 4.
Treating the pointers as arrays (in-line with the struct) – Question graded
out of 6.
```

2) Write the function delete, which removes all of the memory associated with a Market struct (and all of its data), assuming it is all stored on the heap

```
void delete(Market* market) { //3 points
      //1 point per statement. -1 if free(market) comes before other two.
      //Point penalties for extraneous free statements (such as calling free
      //on each item.
      free(market->name);
      free(market->inventory);
      free(market);
}
```

Q4 Trendy (M/F)

Please match each of the following descriptions with the graph below that **best matches**. In each statement, the first quantity mentioned will be on the Y axis, and the second quantity mentioned will be on the X axis. **Assume a linear scale for both the X and Y axes.**

1 point per correct answer (no explanations needed).

I) Cache miss rate versus block size assuming constant cache size.

D. See chart from lecture – block size tradeoff.

II) Speedup versus parallelization of a task with a serial component.

C. Amdahl's law.

III) Speedup versus number of threads while using OpenMP to divide up a computation.

A. Dropoff due to overhead as threads > cores.

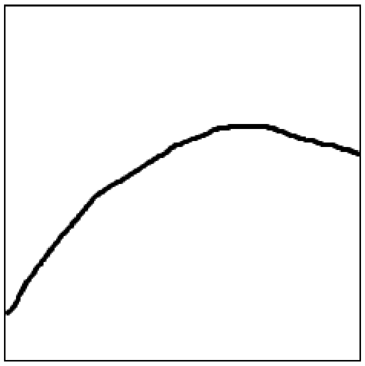
IV) Number of cores per chip versus time.

E. Chart from lecture. No multicore chips until 2004.

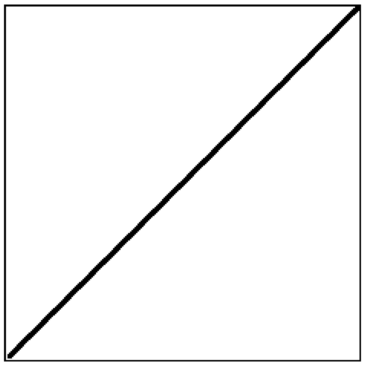
V) Minimal clock period versus number of pipeline stages.

F. Min period = clk-to-q+setup+combinational logic delay. More pipeline stages => less combinational logic delay, other two terms constant.

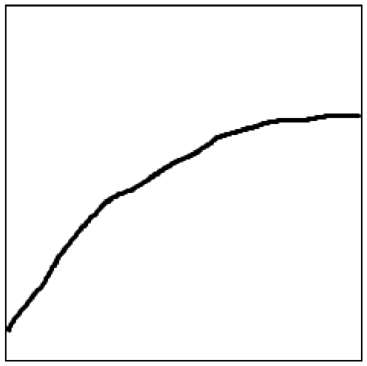
a)



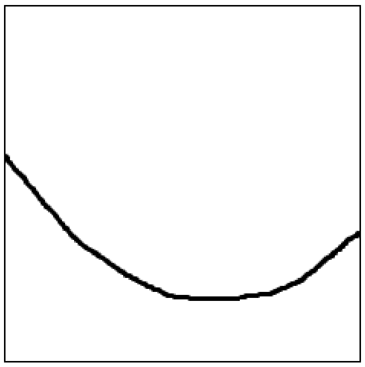
b)



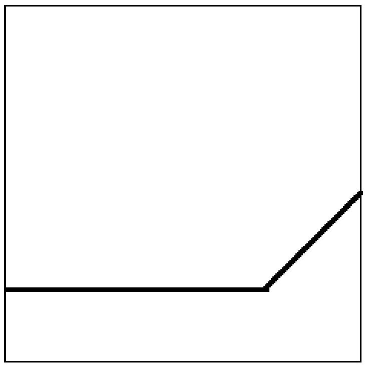
c)



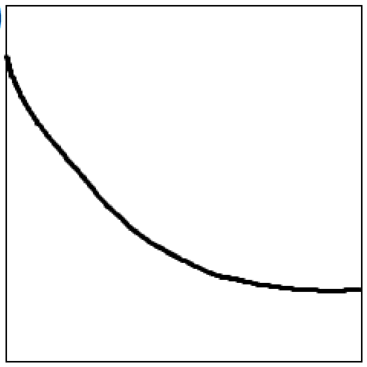
d)



e)



f)



Q4 Trendy (continued)

Please match each of the following descriptions with the graph below that **best matches**. In each statement, the first quantity mentioned will be on the Y axis, and the second quantity mentioned will be on the X axis. **Assume a linear scale for both the X and Y axes.**

I) The digital output of a D Flip-Flop other than during a rising edge.

D. It's constant.

II) Typical CPU power usage versus CPU utilization.

A. Chart from lecture/WSC reading.

III) Power consumption versus clock frequency.

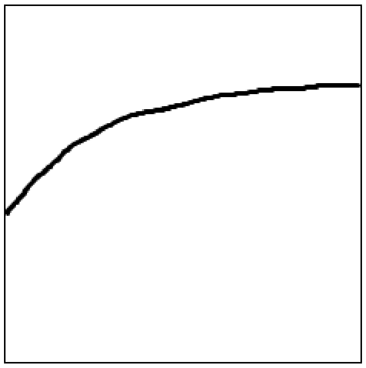
F. $P = CV^2f$.

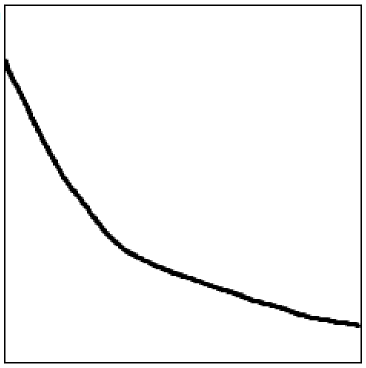
IV) Mean Time to Failure versus number of independent, identical components, assuming no redundant components.

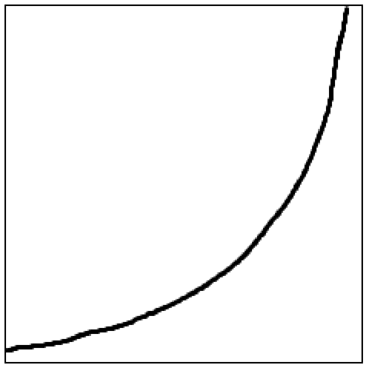
B. MTTF = MTTF of individual component / # components.

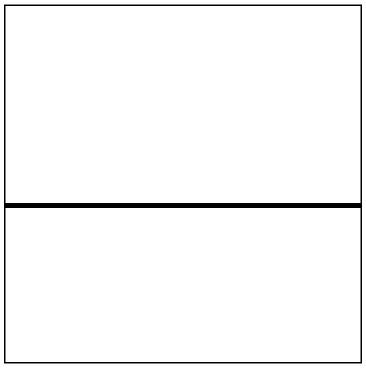
V) Hit Time versus distance from CPU in the memory hierarchy.

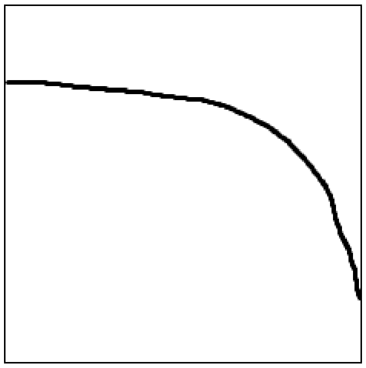
C. Hit time tends to increase super-linearly with each step in the heirarchy (ie L$2 HT / L1$ HT << Disk HT / Mem HT)

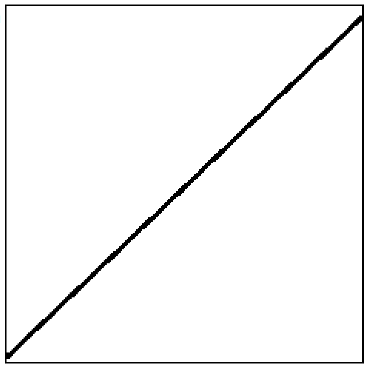
a)


b)


c)


d)


e)


f)


Q5 Encryption (M/F)

The following function applies a simple non-delayed-branch model of encryption. It takes a string as an argument (`$a0`) and an encryption value (`$a1`). It then encrypts the string by adding the encryption value to each char (except the null terminator).

```
encrypt:
      lbu $t0 0($a0)
      beq $t0 $0 end
      addiu $t0 $t0 $a1
      sb $t0 0($a0)
      addiu $a0 $a0 1
      j encrypt
end:  jr $ra
```

You call `encrypt(message, 20)`, where message is a char array of length 2048 (including the null terminator). Your computer has the following specifications:

- L1 TIO = 22|7|3 bits
- L1 Data Cache Hit Time of 1 clock cycle
- Miss Penalty to main memory: 72 clock cycles
- CPI ideal of 1 clock cycle (**single-cycle datapath**)
- 1 GHz clock rate (it's a slow computer)

For the following problems: ignore the instruction `jr $ra`. It will be executed exactly once and thus is negligible. SHOW YOUR WORK.

What is your L1 Data Cache Miss Rate? (2 pt)

1/16

Assuming the cache is initially empty, how many valid cache blocks will be replaced? (2 pt)

128 or 129 both accepted

Instruction Cache blocks are generally 32 bytes or larger. Knowing this, why are instruction cache misses negligible to performance in this function call? (2 pt)

The instruction cache holds the entire program OR

The instruction cache blocks can each hold the entire program

Calculate the average number of **data** memory accesses per instruction. (1 pt)

1/3.

Calculate the number of instructions that will be executed. Leave this as an expression (multiply and addition signs are acceptable)! (1 pt)

6*2047+2

Any answer within +-10 of this was definitely accepted

Calculate the CPU execution time of this code. (2 pt)

CPI=1+(1/3)(1/16)(72)=5/2 Execution Time=(Ins Count)(CPI)(Clock Period)

Any correct application of these formulas was accepted, even if incorrect numbers from previous sections were used. Also, we did not deduct any points for excluding the (Clock Period) term.

Now assume you run the same code on your 5-stage pipelined datapath with **delayed branches and forwarding**.

If you do not reorder the instructions, how many stalls will be necessary per iteration? Specify where (after which exact instructions) stalls are needed, how many, and why. (3 pt)

This question was not very well-posed. The correct interpretation of the pipelining system's operation is that the CPU uses delayed branches to avoid control hazards, but uses hardware stalls to resolve all other hazards. This implies that if "you run the same code on your 5-stage pipelined datapath," the program no longer functions correctly. For this reason, we only graded students on recognizing the data hazard and placing stalls between lbu and beq. We awarded 3pt for recognizing that two stall cycles are needed there, and also awarded 3pt for a more detailed analysis of hardware stall time that takes cache miss time into account. We awarded 2pt for answering that 1 or 3 stalls are needed between lbu and beq

Q6 SIMD (F)

Below is a non-SIMD version of `strncopy`, which copies `n` characters of a string `src` into `dst`.

```
void strncopy(char* src, char* dst, int n) {
        int i;
        for(i = 0; i < n; i++)
                dst[i] = src[i];
}
```

Now SIMDize it using `__m128i` vectors. Feel free to use the following functions:

```
__m128i _mm_loadu_si128 (__m128i *p);
   void _mm_store_si128 (__m128i *p, __m128i a);
```

```
void strncopy(char* src, char* dst, int n) { (5 pt)

        // your code here

        int i;

        for( i = 0 ; i < n / 16 * 16 ; i += 16 )

                _mm_storeu_si128( dst + i , _mm_loadu_si128( src + i ) );

        for( ; i < n ; i++ ) //fringe

                dst[i]=src[i];

}
```

Common Errors

Wrong data size (char's are 1 byte, so 16 chars in one 128 -bit register) (-2 pt)

No fringe case (-1 pt)

Wrong loop variable increment and/or wrong loop bounds (-1 pt)

Dereferenced store and load arguments, (src[i] is a char, src+i is a pointer) (-1 pt)

Missing pointer arithmetic (src+i) (-1 pt)

flipped storeu arguments (no penalty)

What is the max performance improvement factor you could theoretically achieve from using SIMD over a non-SIMD implementation? (2 pt)

16. (Responses that are consistent with their code got credit. 4 was a common answer)

Use Amdahl's law to compute the speedup of the copying operations over 100 characters. (2 pt)

$$Speedup = \frac{1}{(1-F)+\frac{F}{S}} = \frac{1}{(1-.96)+.\frac{96}{16}} = \frac{1}{.04+.06} = \frac{1}{.1} = 10$$

F = 6*16/100 = 96/100 (fraction of improved operations)   S = 16 (speedup on improved fraction)

Common Errors

Some people tried to use (Old Execution Time / New Execution Time). Unless you applied Amdahl's law (like the question asked), you wouldn't have gotten the correct answer.

Some people confused what values F and S should be set to.

Some people tried changing their fraction to accommodate for the for loops and other code. The question asked for the speedup of the copying operations, which made the math much simpler, but if it was clear exactly how you accommodated for this and you didn't make any other mistakes, you got credit.

Q7 Thread Level Parallelism (F)

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`.

2 points each: 1 for the circled issue and 1 for the justification. No credit given if justification missing.

```
//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]

Solution:

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.

Also accepted:

Sometimes incorrect – variable `i` declared outside of parallel section, so each thread accesses the same variable (not true in practice). This can cause iterations of `i` to be skipped if consecutive `i++` calls are made by different threads. Still possible to alternate storing and incrementing across all threads, so can still be correct sometimes.

```
//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

Solution:

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.

```
//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

Solution:

Always incorrect – 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.

Also accepted:

Sometimes incorrect – This code will execute correctly for `len < 4`. Otherwise, it executes incorrectly for the reasons listed above.

Q8 Can you Unlock this Problem (F)

We want to implement the simplest MIPS atomic lock, where the lock can only be 0 for unlocked and 1 for locked, in a test-and-set manner.

The MIPS function `lock` is shown below, assuming the address of the key is stored in register `$s0`.

```
lock: ll   $t0, 0($s0)
      bne  $t0, $0, lock
      addi $t0, $0, 1
      sc   $t0, 0($s0)
      beq  $t0, $0, lock
```

a) We would like to visualize the use of this lock as a finite state machine. The three states will be `test` (`0b00`), `set` (`0b01`), and `write` (`0b10`). `Test` and `set` are for requesting the lock, and `write` is for when you are allowed to write to the critical section.

In `test`, the 1-bit input is the lock's value.

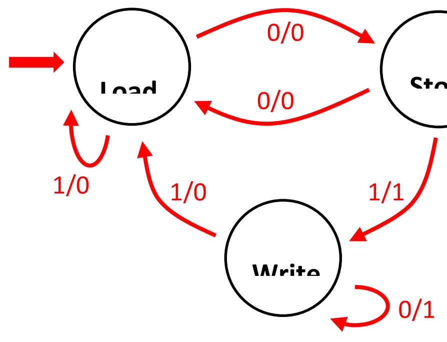
In `set`, the input is the result of store conditional's test.

In `write`, the input is whether or not you are done with the lock (1 means done).

The output will be a 1 if you hold the lock and 0 if you don't.

Additionally, assume that once you are done with the lock, you immediately try to grab it again.

Draw the FSM and fill in the rest of the logic table below:



CS1	CS0	In	NS1	NS0	Out
0	0	0	0	1	0
0	0	1	0	0	0
0	1	0	0	0	0
0	1	1	1	0	1
1	0	0	1	0	1
1	0	1	0	0	0
1	1	0	X	X	X
1	1	1	X	X	X

4 points total for FSM. -0.5 for wrong/no initial state, -0.5 for small errors in input/output, -1 for errors in input/output that demonstrate a lack of understanding of the MIPS lock.

1 point (all-or-nothing) for each column of the LUT if it matched your FSM. Don't cares (X's) could be left blank or filled in if it was consistent with your part b answer.

b) Write out the Boolean expression for NS1. Full credit for the most simplified expression, -1 for any other valid solution.

Unsimplified answer (+1): $\mathrm{NS1} = \overline{\mathrm{CS1}} \cdot \mathrm{CS0} \cdot \mathrm{In} + \mathrm{CS1} \cdot \overline{\mathrm{CS0}} \cdot \overline{\mathrm{In}}$

Simplified answer (+2): $\mathrm{NS1} = \mathrm{CS0} \cdot \mathrm{In} + \mathrm{CS1} \cdot \overline{\mathrm{In}}$

Q9 Pipelined Single Memory Datapath (F)

The diagram on the next page shows a simplified and incomplete 2-stage datapath. The stages are Instruction Fetch and Execute (Decode, Execute, Memory, Write Back). Unlike normal datapaths though, **THIS ONE HAS ONLY ONE MEMORY BLOCK**. This is naturally problematic for load and store instructions, because they conflict with instruction fetch.

Decode, Register File, and the ALU all work as intended.

New Signals and Buses

- Data Address (which has the same value as ALU Result) holds the target address for a load or store instruction. In other words, if I'm executing `sw rt, imm(rs)`, Data Address = `imm+R[rs]`.
- Data for Store holds the data to be stored for a store instruction. In other words, if I'm executing `sw rt, imm(rs)`, Data for Store = `R[rt]`.
- Store? holds 1 if the instruction being decoded is a store instruction, 0 otherwise.
- Load? holds 1 if the instruction being decoded is a load instruction, 0 otherwise.
- Data Access? is the value obtained from OR-ing Store? and Load?

What kind of hazard do we get from having a single memory block?

Structural Hazard. 2 stages competing for the same Memory block. (1 pt)

We've decided to resolve this hazard by use of stalls. Describe exactly when we need to stall.

After a load or store instruction. (1 pt)

Implement the following signals by adding necessary datapath elements:

1) Stall should be set to 1 whenever a stall should happen.
2) Enable PC+4 should be set to 1 whenever PC should be incremented by 4 on the next rising clock edge. You may assume that this datapath does not support jumps or branches.

All standard datapaths and control signals are present.
All state elements already have a clock input.

Enable PC+4 (3 pt)

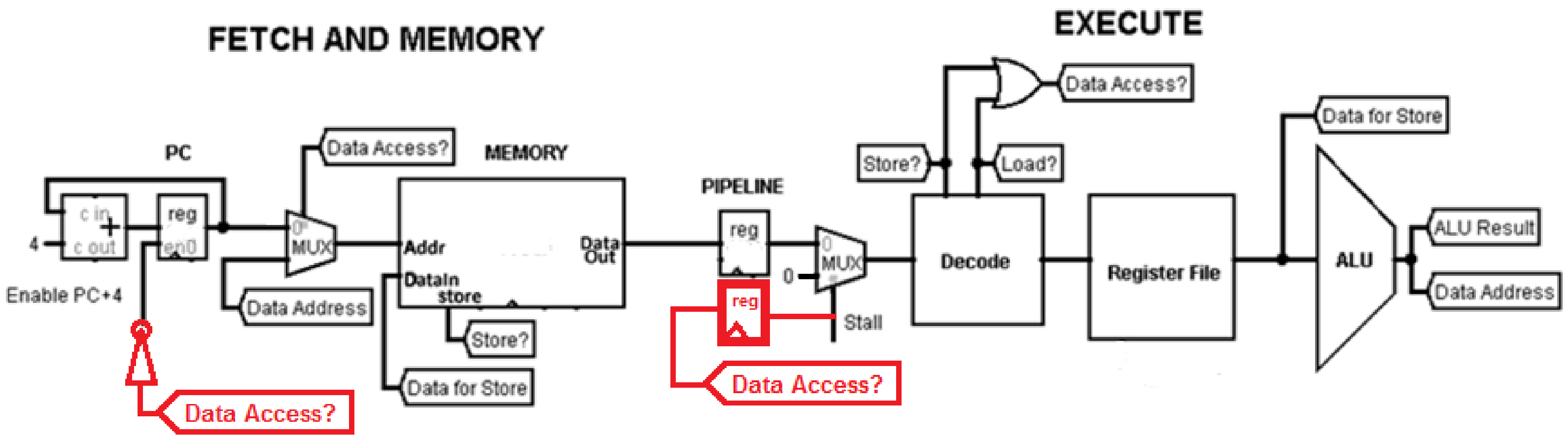
1 pt for the correct signal (OR-ing together Store? and Load? was accepted. We just did it for you with the Data Access? signal)

2 pt for the NOT gate (some people used muxes. Those were accepted)

Stall (5 pt)

1 pt for the correct signal

4 pt for the register. The register is needed to stall the correct instruction (not the current instruction being executed, which would be the load or store). It's also needed to make the circuit stable (the signal coming out of the MUX determines the value for Data Access? and the Data Access? signal determines what comes out of the MUX).



Q10 Virtual Memory (F)

In this problem, we are running two different processes on our computer.

Our system has the following properties:

- 1 MiB of Physical Address Space – 20-bit physical addresses (PA)
- 4 GiB Virtual Address Space – 32-bit virtual addresses (VA)
- 32 KiB page size – 15-bit page offset
- 4-entry fully-associative TLB, LRU replacement
- Page tables (PTs) use write-back policy with permission bits for read (rd), write (wr), and execute (ex).

a) Numbers – Fill in the blanks with the appropriate numbers:

32-15=17	# tag bits for TLB	___20___	bit-width of PT address register
2^{20-15}=32	max # of valid entries in a PT	_5+5=10_	bit-width of an entry in a PT
___2____	# of PTs		(there are 5 extra bits)

1 point each: fully-associative TLB means no index bits, PT sits in physical mem (PT address is PA), PT valid entries limited by physical mem capacity, PT only holds PPN (5 bits), one PT per process.

b) Below is an excerpt from one of the processes (Process 1), which uses a large square matrix of 32-bit integers. What is the largest gap between successive memory accesses (in the virtual address space) in **bytes** taken in the `for` loop?

```
#define MAT_SIZE = 2048
for(int i=0; i<MAT_SIZE; i++)
      mat[i*(MAT_SIZE+1)] = i;
```

`2049*4=8196B`

1 point, stride length is `MAT_SIZE+1` integers, which is `4*(MAT_SIZE+1)` bytes.

c) Assume that the matrix `mat` is stored contiguously in memory and that `mat[0]` is at the beginning of a page. Assuming that Process 1 is the only process running, calculate the following hit rates (HRs) for the first execution of the `for` loop: (2 points each, +1 for TLB miss rate)

_____0%_____ PT Hit Rate ____75%_____ TLB Hit Rate

We access memory in a strictly increasing manner: `(i+1)*(MAT_SIZE+1) > i*(MAT_SIZE+1)`. Because of this, we never revisit a page once we have left it. TLB hits do NOT count as PT hits, so the PT hit rate is 0%.

The square matrix has rows and columns of size $4 \times 2^{11} = 2^{13}$ B. Regardless of whether it is stored as row-major or column-major, a page holds $2^{15-13} = 4$ rows/columns of the matrix. The memory access pattern is the diagonal elements of the matrix (one access per row/column), so we miss once every 4 accesses and our hit rate is 3/4 = 75%.

Q11 Who Invited Those People Again? (M)

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
```

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

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

15 is stored to $t7 (both accepted)

For each of the following questions, **CIRCLE** either a, b, c, d, or e.

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.`

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.`

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=0x000000DC`

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).