Computer Systems Organization
CSCI-UA.0201 Spring 2021
Final Exam ANSWERS

1) (10 points) True/False.

a) T An assembler translates assembly code into machine code.

b) T In C, if p is declared as an "int *" and has the value 1000 decimal, then p+10 will compute the value 1040 decimal.

c) T In assembly, a function F can modify a caller-saved register without worrying that it is corrupting a value needed by the function that called F.

d) F In assembly, a function F can modify a callee-saved register without worrying that it is corrupting a value needed by the function that called F.

e) F In C, if x is an integer variable and `MASK` is the constant 0x1, then "`(x & MASK == 0)`" will be true if x is 0.

f) T In assembly, you cannot always use the 64-bit registers (%rax, %rbx, etc.) and 64-bit instructions (addq, cmpq, etc.) rather than the corresponding 32-bit registers (%eax, %ebx, etc.) and 32-bit instructions (addl, cmpl, etc.).

g) F A DRAM cell needs to be periodically refreshed because DRAM (e.g. main memory) is volatile memory, so it loses its data when the computer's power is turned off.

h) T In the memory hierarchy discussed in class, the larger memories in the hierarchy tend to be slower than the smaller memories in the hierarchy.

i) T A multiplexer with 2^N inputs has N select bits (wires) because N bits can take on a value between 0 and 2^N-1

j) T On a machine which uses two's complement to represent negative numbers, performing an arithmetic (not logical) shift right by one on a negative number will insert a 1 into the new leftmost bit.

2) (10 points: (a) 4pts, (b) – (d) 2 points) Fill in the blanks.

a) Write the number F3A5 hex in binary: 1111 0011 1010 0101.

b) log 128M = 27 .

c) In order to access all bytes in a 128G memory, an address must have at least 37 bits.

d) If an integer variable is represented by 33 bits, how many different numbers could that variable take on? 8G

3) (10 points: (a) 3pts, (b) 4pts, (c) 3pts)

a) Fill in the blank so that the following code prints "abc".

```
int x = 0xabcdef;
printf("%x", x >> 12 );
```

b) Fill in the blank so that the following code prints "15".

```
int a[] = {10, 15, 20, 25, 30, 35, 40};
int *p = &(A[1]);    // or int *p = A+1;
printf("%d\n", *p);
```

c) What will %rax contain after executing the following assembly code? If it is impossible to know given the information, or the code is not valid or crashes the program, indicate that.

```
movq $10,%rcx
leaq (%rbx,%rcx,8),%rax
subq %rbx,%rax
```

Answer: 80

4) (20points: (a) 2pts (b) 3pts (c) 15pts) Given the following declaration in C,

```
typedef struct node {
      int A[30];
      struct node *left;
      struct node *right;
} NODE;
```

answer the following questions:

a) Since the C compiler sometimes puts empty space between the fields of a node (due to alignment requirements), if you needed to know the offsets of the start of the `A`, `left`, and `right` fields from the start of a NODE, you could write some C code to print out those offsets. Fill in the blanks of the code below to do so:

```
NODE *p = malloc(sizeof(NODE));
printf("Offsets:  A = %ld, left = %ld, right = %ld\n",
       p->a - p,   &(p->left) - p,   &(p->right) - p);
```

For example, if the `A` field starts at 36 bytes after the start of the NODE, the first part of the printout would be "Offsets: A = 36". Don't worry about performing casts to get the types right.

b) If the `A`, `left`, and `right` fields were, in fact, contiguous (no space in between the fields), then what would the offset of each field be from the beginning of a node? Just write a number into each blank.

A: 0 , left: 120 , right: 128

c) Assuming the fields of a NODE are contiguous, and given your answer to the previous question about the offsets of the fields, translate the following C function, `copy_to_left`, into x64 assembly:

```
void copy_to_left(NODE *p) {
    if ((p != NULL) && (p->left != NULL)) {
       memcpy(p->left->A, p->A, 40);
    }
}
```

where the built-in function `memcpy` takes 3 parameters: (1) destination address, (2) source address, and (3) a number of bytes, and copies that number of bytes from the destination address to the source address. As you can see, `copy_to_left` takes a NODE pointer `p` as a parameter and, if neither `p` nor `p->left` are NULL, it copies the elements of `p->A` to `p->left->A`. You do not need to write the code for `memcpy`, just call it. The entire code should be around 15 lines or so. Note: The calling conventions for MacOS/Linux and Windows are found on the last page of this exam.

Answer: This is for MacOS. Linux would be identical except for no underscore before names.

```
_copy_to_left:
      pushq %rbp
      movq  %rsp, %rbp
      movq  %rdi,%rsi        # put p in %rsi
      cmp   $0,%rsi          # if p == NULL,
      je    DONE             #   jump to DONE
      movq  120(%rsi),%rdi   # put p->left in %rdi
      cmp   $0,%rdi          # if p->left == NULL
      je    DONE             #   jump to DONE
      # p->left is already in %rdi, p is already in %rsi,
      # so just pass 40
      movq  $40,%rdx         # passing size (40)
      call  _memcpy          # calling memcpy
      # just return
DONE:
      popq  %rbp
      retq
```

This is for Window/Cygwin.

```
copy_to_left:
        pushq   %rbp
        movq    %rsp, %rbp
        movq    %rcx,%rdx       # p in %rdx
        cmp     $0,%rdx         # if p == NULL,
        je      DONE            #   jump to DONE
        movq    120(%rdx),%rcx   # p->left in %rcx
        cmp     $0,%rcx         # if p->left == NULL
        je      DONE
        # p->left is already in %rcx, p is already in %rdx,
        # so just pass 40
        movq    $40,%r8     # passing size (40)
        subq    $32,%rsp     # this is mandatory
        call    memcpy       # calling memcpy
        addq    $32,%rsp     # mandatory
```

```
        # just return
DONE:
        popq    %rbp
        retq
```

5) (20 points: (a) – (c) 5pts each, (d) 3pts, (e) 2pts)

a) Suppose your desktop computer has a processor which has a separate instruction cache (I-cache) and data cache (D-cache) and, on a cache hit, can execute an instruction every cycle. Suppose also that the penalty for a cache miss is 100 cycles. Given a program that executes N instructions overall, with a 3% instruction cache miss rate and a 2% data cache miss rate, where 30% of all instructions need to access data in memory, how many cycles does the program take to execute? Show your work (as I did in class).

Answer:

Total cycles = N + (0.03 * N * 100) + (0.02 * N * 0.3 * 100)
= N + 3N + 0.6N
= 4.6N

b) Assuming that you have some money to upgrade your computer with a new processor and you have a choice of a new processor that doubles the size of both the I-cache and D-cache (so assume that the miss rate is halved), or a new processor that triples the size of the I-cache (so assume there is 1/3 of the original I-cache miss rate) but leaves the D-cache size the same, which processor would you pick? Justify your answer by showing which would take fewer cycles to execute the above program.

Answer:

Cutting the I-cache and D-cache miss rate in half, results in halving of both penalty terms:

Total cycles = N + 1.5N + 0.3N
= 2.8N

Reducing the I-cache miss rate to 1/3 of its prior value divides the instruction penalty term by 3:

Total cycles = N + N + 0.6N
= 2.6N

It's close, but tripling the size of the I-cache is better in this case.

c) Suppose that a processor has 32-bit addresses and 32-bit words, a 8-word cache line, and an 2MB 4-way set associative cache (i.e. the cache has a capacity of 2MB of actual data). Indicate how an address would be partitioned into fields to allow a single word to be fetched from the cache by the CPU (just indicate what the fields are, the order of the fields in the address, and how many bits each field is).

Answer:
32-bit words = 4 bytes/word, so byte offset is lowest 2 bits.
8 words/cache line, so word offset is the next 3 bits.
2MB cache = 2^{21} bytes
= 2^{19} words

= 2^{16} cache lines (since there are 2^3 words per line)
= 2^{14} sets (since there are 2^2 lines per set)

so the set index is the next 14 bits.
The tag is the remaining 13 bits.

d) There are three categories of cache misses, according to why they occur. Indicate what those categories are named and give one sentence describing each of them.

Answer:
Compulsory misses: These are misses that occur the first time a cache line is accessed, so the cache line won't be in the cache yet.

Conflict misses: These are misses that occur when multiple cache lines being used by a program map to the same index, so -- particularly in a direct-mapped cache – those lines can't exist in the cache at the same time.

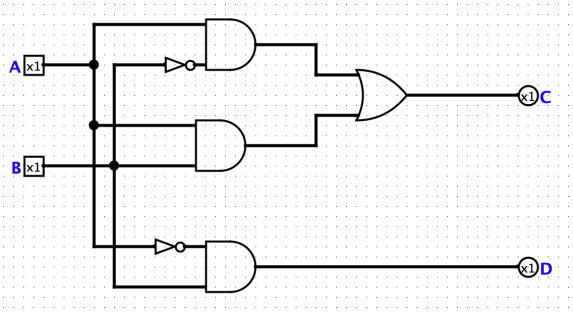
Capacity misses: These are misses due to the cache not being large enough to hold the cache lines a program currently needs (the "working set").

e) For which one of the categories of cache misses in the previous question is the cache miss rate reduced by using a set-associative cache rather than a direct-mapped cache? Briefly explain.

Answer:
Conflict misses are reduced, since cache lines that would otherwise conflict can be stored within the same set of the cache.

6) (20 points: (a) 3, (b) 5, (3) 2 (d) 10) For these questions, you can assume that AND and OR gates can take any number of inputs (not just two).

a) Write out the truth table corresponding to the following circuit (the "x1" you see just means 1 bit).

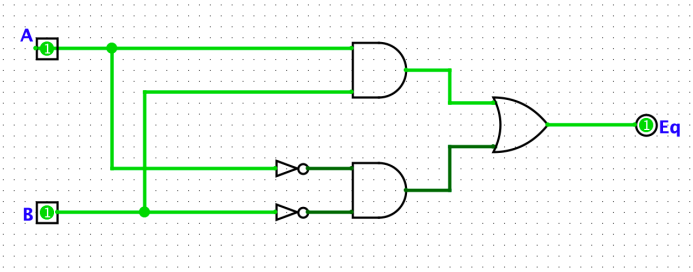


Answer:

A	B	C	D
0	0	0	0
0	1	0	1
1	0	1	0
1	1	1	0

b) Draw, or describe in words, a circuit that implements 1-bit equality. That is, the circuit should have two 1-bit inputs, A and B, and one 1-bit output Eq, such that Eq is true if A and B have the same value, otherwise Eq is false. Only use AND, OR, and NOT gates.

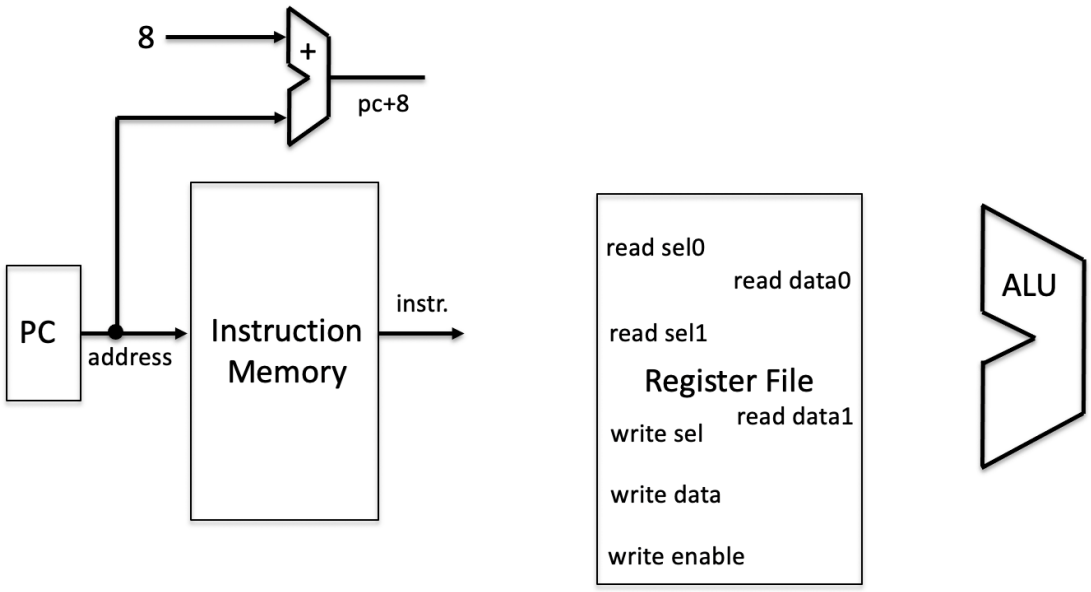
Answer:



c) Describe by a drawing or by words how you would implement a circuit for a 64-bit equality operator, using your 1-bit equality operator above. The 64-bit version should take two 64-bit inputs, A and B, and produce a 1-bit output Eq that is true if A and B have the same value, otherwise Eq is false.

Answer: Send each bit of A and the corresponding bit of B to an equality circuit, for a total of 64 equality circuits. Take the 1-bit outputs of all 64 equality circuits and send them to a (huge) AND gate. The output of the AND gate is the Eq output.

d) Below, for your reference is a diagram that shows pieces of the processor datapath discussed in class (where the program counter, "PC", is the %rip register on the x64).



As discussed in class, the `cmpq` instruction simply performs a subtraction and remembers whether the result is positive, zero, or negative by setting certain bits in a flag register. Suppose, however, for the purposes of this exam, `cmpq` just saves the result of the subtraction in register 31. Describe in drawing or words, what you would add to the above datapath diagram to implement the `je` (jump if equal) instruction (not the `cmpq` instruction). Note that the assembler converts the instruction

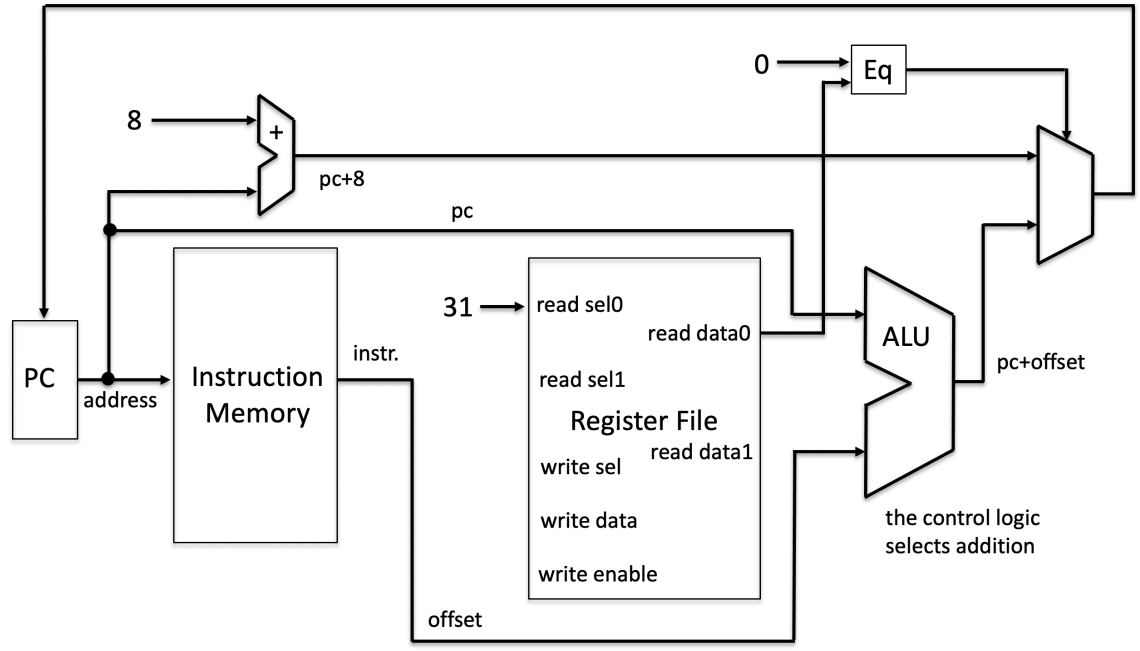
```
je  LABEL
```

into

```
je offset(%rip)
```

so that `offset(%rip)` is the address to jump to (corresponding to where the label is in the program). The machine code for the `je` instruction contains just two fields: the op code and the offset. Feel free to use any circuit discussed in class, as well as your circuit for computing 64-bit equality from the previous question. Also, be sure to show (or describe) what wires are used to connect the various components.

Answer:



X86-64 Calling Conventions

Unix (e.g. macOS and Linux)

Passing Parameters:

- First six integer/pointer parameters (in order): %rdi, %rsi, %rdx, %rcx, %r8, %r9
- Additional parameters are passed on the stack, pushed in right-to-left (reverse) order.

Return Value:

- %rax (for 64-bit result), %eax (for 32-bit result)

Caller-Saved Registers (may be overwritten by called function):

- %rax , %rcx , %rdx , %rdi , %rsi , %rsp , %r8, %r9, %r10, %r11

Callee-Saved Registers (must be preserved – or saved and restored – by called function):

- %rbx, %rbp, %r12, %r13, %r14, %r15

Microsoft (e.g. for Windows)

Passing Parameters:

- First four integer/pointer parameters (in order): %rcx, %rdx, %r8, %r9
- Additional parameters are passed on the stack, pushed in right-to-left (reverse) order.
- Important: The caller must allocate 32 bytes on stack (by subtracting 32 from %rsp) right before calling the function. Don't forget to restore the %rsp (by adding 32) after the call.

Return Value:

- %rax (for 64-bit result), %eax (for 32-bit result)

Caller-Saved Registers (may be overwritten by called function):

- %rax, %rcx, %rdx, %r8, %r9, %r10, %r11

Callee-Saved Registers (must be preserved – or saved and restored – by called function):

- %rbx, %rbp, %rdi, %rsi, %rsp, %r12, %r13, %r14, %r15