Prelim 2
Computer Science 3410, Cornell University
Prof. Weatherspoon 30 April 2015

Solutions

Read all of the following information before starting the exam:

Please write your name and NetId/email on the exam **now**.

This is a **closed book and notes** examination: *NO BOOK, NOTES, CALCULATORS, OR CELL PHONES*. You have **120 minutes** to answer as many questions as possible.

There are 6 problems on this exam. It is 33 pages long; make sure you have the whole exam. You will likely find some problems easier than others; read **all** problems before beginning to work, and use your time wisely. The exam is worth 100 points total.

Before starting the exam, write your name and netid on all pages of the exam.

Do all written work on the exam itself. If you are running low on space, write on the back of the exam sheets and be sure to write (OVER) on the front side. It is to your advantage to show your work — we will award partial credit for incorrect solutions that are headed in the right direction. If you feel rushed, try to write a brief statement that captures key ideas relevant to the solution of the problem.

Make your answers as concise as possible. If a question is unclear, please simply answer the question and state your assumptions clearly. If you believe a question is open to interpretation, then please ask us about it!

Problem	Points	Score
1	1	
2	9	
3	23	
4	22	
5	20	
6	25	
Total	100	

1. Write your name and NetID [1 pt]

[This page intentionally left empty]

2. Short Answers [9 pts] (parts a–c)

   Be brief. Answer should be one or two sentences at most. Verbose answers will lose points.

   (a) [3 pts] TRUE/FALSE. A `Privileged Mode` is necessary to enforce protection between processes. Justify answer.

   Answer:

   TRUE. Otherwise, a processes could write to any location in memory, could run as long as it wants, and could otherwise take over the entire computer. A privileged mode is necessary multiplex a computer while protecting processes from one another and to prevent the above misbehaviors since only a process running in privileged mode (the operating system) can access memory that may be marked non-readable, can set a timer, can set the page table base register that defines an address space, etc.

   Grading Rubric:
   +1.5 for correct answer
   +1.5 for justification

   (b) [3 pts] TRUE/FALSE. Increasing the cacheline size always increases performance (i.e. increases hit rate). Justify answer.

   Answer:

   FALSE. The hit rate may actually decrease (similarly, the miss rate may actually increase) since fewer cachelines will be available; as a result, a cacheline may need to be replaced before any of the benefits of spatial locality are reaped. Furthermore, the cost of a miss (miss penalty) will increases since a large block of data needs to be fetched from the next level in the memory hierarchy.

   Grading Rubric:
   +1.5 for correct answer
   +1.5 for justification

   (c) [3 pts] List/describe three advantages of virtual memory.

   Answer:

   Process can be larger than physical memory.
   Processes can share memory.
   Can set different permissions on different parts of the address space.
   Can multiplex a processor between different processes given each process the illusion that it owns all of memory.

   Grading Rubric:
   +1 for each benefit

   -1 for an incorrect benefit

3. Caches and Memory Hierarchy [23 pts] (parts a–f)

Assume that we have a 32-bit processor (with 32-bit words) and that this processor is byte-addressed (i.e. addresses specify bytes). Suppose that it has a 512-byte cache that is two-way set-associative, has 4-word cachelines, and uses LRU replacement. Split the 32-bit address into "tag", "index", and "cacheline offset" pieces.

(a) [3 pts] Which address bits comprise each piece?

tag:
index:
cacheline offset: bits 3-0 (Given)

Answer:
tag:*bits 31-8*
index:*bits 7-4*
cacheline offset: bits 3-0 (Given)

Grading Rubric:
+1.5 points each

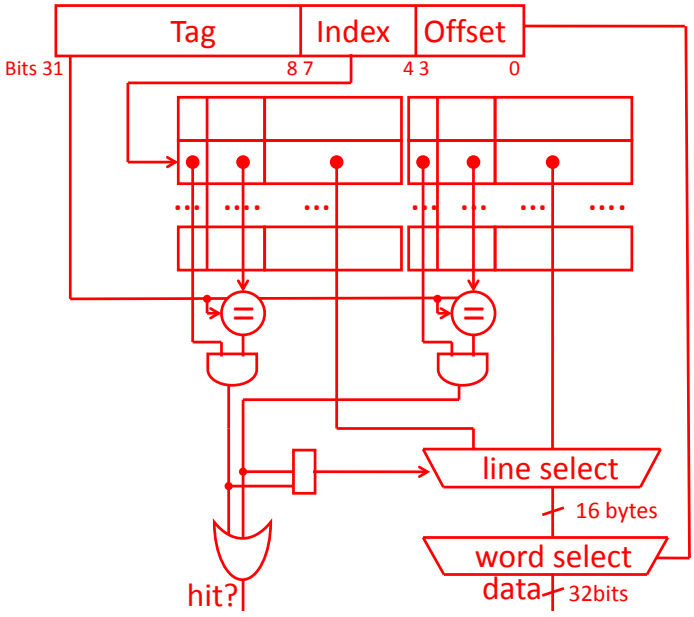
(b) [3 pts] How many sets does this cache have? Explain.

Answer:
4 bits for index ⇒ 16 sets

Grading rubric:
+3 right answer and explanation.

(c) [4 pts]

Draw a block diagram for this cache. Show a 32-bit address coming into the diagram and a 32-bit data result and Hit signal coming out.
Include, all of the comparators in the system and any muxes as well. Include the the tag matching logic, and any muxes, etc.



Answer:

Grading Rubric:
Started with 4 points,then subtracted for the following reasons:
-1 not two way associative
-1 no hit? bit
-1 no data output
-1 lacking or incorrect use of muxes
-1 incorrect equality checking
-1 other inconsistencies

(d) [6 pts]

Below is a series of memory read references set to the cache from part (a). Assume that the cache is initially empty and classify each memory references as a hit or a miss. Identify each miss as either **compulsory**, **conflict**, or **capacity**.

Byte Address	Hit/Miss	Miss Type?
0x300		
0x1BC		
0x206		
0x109		
0x308		
0x1A1		
0x1B1		
0x2AE		
0x3B2		
0x10C		
0x205		
0x301		
0x3AE		
0x1A8		
0x3A1		
0x1BA		

Answer:

	Byte Address	Hit/Miss	Miss Type?
0011 0 **000** 0000	0x300	M	Compulsory
0001 1 **011** 1100	0x1BC	M	Compulsory
0010 0 **000** 0110	0x206	M	Compulsory
0001 0 **000** 1001	0x109	M	Compulsory
0011 0 **000** 1000	0x308	M	Conflict
0001 1 **010** 0001	0x1A1	M	Compulsory
0001 1 **011** 0001	0x1B1	H	—
0010 1 **010** 1110	0x2AE	M	Compulsory
0011 1 **011** 0010	0x3B2	M	Compulsory
0001 0 **000** 1100	0x10C	H	—
0010 0 **000** 0101	0x205	M	Conflict
0011 0 **000** 0001	0x301	M	Conflict
0011 1 **010** 1110	0x3AE	M	Compulsory
0001 1 **010** 1000	0x1A8	M	Conflict
0011 1 **010** 0001	0x3A1	H	—
0001 1 **011** 1010	0x1BA	H	—

Grading Rubric:
Started with 6 points, then subtracted 0.5 points for each incorrect row.

(e) [3 pts] Calculate the miss rate and hit rate?

Answer:

Hit Rate = 4/16 = 0.25

Miss Rate = 1 - Hit Rate = 12/16 = 0.75

Grading Rubric:
Graded based on answer to part 3(d)
+1.5 each

(f) [4 pts] You have a 1 GHz processor (i.e. 1 clock cycle per ns) with 2-levels of cache, 1 level of DRAM (memory), and a DISK for virtual memory. Assume that it has a Modified Harvard architecture (separate instruction and data cache at level 1). Assume that the memory system has the following parameters:

Component	Hit Time	Miss Rate	Block Size
First-level Cache	1 cycle	4% Data 1% Instruction	64 bytes
Second-level Cache	20 cycles + 1 cycle/64 bits	2%	128 bytes
DRAM (Memory)	100ns + 25ns/8 bytes	0.1%	16K bytes
DISK	50ms + 20ns/byte	0%	16K bytes

Finally, assume that there is a TLB that misses 0.1% of the time on data (it does not miss on instructions) and which has a fill penalty of 40 cycles. What is the average memory access time (AMAT) for Instructions? For Data (assume all reads)?
Write down the general formula for AMAT.

Answer:

$$AMAT = \%Hit\ x\ HitTime + \%Miss\ x\ MissTime$$
$$where\ MissTime = HitTime + Miss\ Penalty$$

$$So,\ AMAT = HitTime + \%\ Miss\ x\ Miss\ Penalty$$

$AMAT_{DISK} = (5 \times 10^7 \text{ns}) + (16384 \times 20) = 50,327,680\text{ns} = 50,327,680$ *cycles*
$AMAT_{DRAM} = (100\text{ns} + 25\text{ns} \times 16) + 0.001 \times AMAT_{DISK} = (500\text{ns} + 50327.680\text{ns}) = 50,827.680\text{ns} = 50,827.680$ *cycles*
$AMAT_{L2} = (20 + 8) + 0.02 \times AMAT_{DRAM} = 1044.5536$ *cycles*

$AMAT_{INST} = (1 + 0.01 \times AMAT_{L2}) = 11.445536$ *cycles*

$AMAT_{DATA} = (1 + 0.04 \times AMAT_{L2} + 0.001 \times 40) = 42.822144$ *cycles*

Rounding
$AMAT_{DISK} = (5 \times 10^7 \text{ns}) + (16000 \times 20) = 50,320,000\text{ns} = 50,320,000$ *cycles*
$AMAT_{DRAM} = (100\text{ns} + 25\text{ns} \times 16) + 0.001 \times AMAT_{DISK} = (500\text{ns} + 50320\text{ns}) = 50,820\text{ns} = 50,820$ *cycles*
$AMAT_{L2} = (20 + 8) + 0.02 \times AMAT_{DRAM} = 1034.4$ *cycles*

$AMAT_{INST} = (1 + 0.01 \times AMAT_{L2}) = 11.344$ *cycles*

$AMAT_{DATA} = (1 + 0.04 \times AMAT_{L2} + 0.001 \times 40) = 42.416$ *cycles*

Grading Rubric:
+1 Formula
+1 Recursively applying formula
+1 Applying TLB correctly to $AMAT_{DATA}$
+1 $AMAT_{INST}$ *and* $AMAT_{DATA}$

[This page intentionally left empty]

4. MMU and Virtual Memory [22 pts] (parts a–i)

64-bit computers are *very* common. We want to understand what 64-bits really means for the virtual memory system. As a result, in this problem, we consider a byte addressable virtual memory system with **64-bit virtual addresses**, **48-bit physical addresses** and **16 kB pages**.

Write answers using *both* **binary** and **decimal** notation.
(For example, for 2^{14} *bytes, we could write both 16 kB and 16 thousand-bytes)*

For *binary notation*, you may write the amount of bytes using k for kilo-, M for mega-, G for giga-, T for tera-, P for peta-, E for exa-, Z for zetta-, and Y for yotta-byte: kB, MB, GB, TB, PB, EB, ZB, YB, respectively.

For *decimal notation*, you may approximate and just write thousand-, million-, billion-, trillion-, quadrillion-, quintillion-, sextillion-, septillion-, octillion-bytes (. . . googol-bytes); or 10^3, 10^6, 10^9, 10^{12}, 10^{15}, 10^{18}, 10^{21}, 10^{24}, 10^{27} (. . . 10^{100}), respectively.

(a) [2 pts] What is the maximum amount of physical memory that the system could support?

Answer:

$2^{48} = 256$ *TB*
Also, $256 \times 10^{12} = 256$ *trillion bytes*

Grading Rubric:
+1.5 for the correct answer in binary or decimal
+2 for the correct answer in both

(b) [2 pts] What is the maximum size of a single process; i.e. what is the size of the virtual memory space for a single process?

Answer:

$2^{64} = 16$ *EB*
Also, $16 \times 10^{18} = 16$ *quintillion bytes*

Grading Rubric:
+1.5 for the correct answer in binary or decimal
+2 for the correct answer in both

(c) [3 pts] If this system used a *single-level* page table, how many entries would it need to have? Why might this be a problem?

Answer:

2^{50} *entries*
Also, $10^{15} = 1$ *quadrillion entries*

1 quadrillion page table entries is ridiculously large.

Grading Rubric:
+1.5 for the correct answer in binary or decimal
+2 for the correct answer in both +1 for correct explanation

(d) [3 pts] If all entries in the *single-level* page table took up 8 bytes each, what is the minimum amount of physical memory that a 2 MB process would occupy (if the entire process and page table was in memory)?

Answer:

$2^{50} \times 8 + 2$ *MB* =
$2^{53} + 2$ *MB* =
8 PB + 2 MB

Also, 8 quadrillion bytes plus 2 million bytes

Grading Rubric:
+2.5 for the correct answer in binary or decimal
+3 for the correct answer in both

(e) [3 pts] How many entries would the **first level** of a *two-level* page table have, assuming that the same number of bits are used for both levels?

Answer:

2^{25} *entries*
Also, $32 \times 10^6 = 32$ *million entries*

Grading Rubric:
+2.5 for the correct answer in binary or decimal
+3 for the correct answer in both

(f) [3 pts] If all entries in the *two-level* page table (both first and second level) took up 8 bytes, what is the minimum amount of physical memory that a 2 MB process would occupy (if the entire process and page tables were in memory)?

Answer:

first-level + second-level + process =
$(2^{25} \times 8) + (2^{25} \times 8) + 2$ *MB =*
$2^{25} * 2^3 + 2^{25} * 2^3 + 2$ *MB =*
$2^{28} + 2^{28} + 2$ *MB=*
$2^{29} + 2$ *MB=*
$512\text{MB} + 2$ *MB=*
514 *MB*
Also, $514 \times 10^6 = 514$ *million bytes*

Grading Rubric:
+2.5 for the correct answer in binary or decimal
+3 for the correct answer in both

(g) [2 pts] Why is a page table entry 8 bytes, instead of 4. Briefly, explain in *one sentence*. More than one sentence or run-on sentences will lose points.

Answer:

A page table entry (PTE) contains a physical page number (ppn) which needs to be at least 34 bits (=48 bits physical memory - 14 bits page size = 34 bits), where 34-bits s over 4 bytes. Further, a PTE contains valid, dirty, and permission bits.

Grading Rubric:
-1 for not mentioning physical page number and its size.

(h) [2 pts] What is the problem of 64-bit virtual memory address spaces (based on your answers above)?

Answer:

The overhead is way too high!

Grading Rubric:
+2 for a correct explanation.

(i) [2 pts] If instead of two-level page tables, we had *five*, what are some other issues with 64-bit virtual memory address spaces?

Answer:

Slow to translate virtual to physical addresses. It would take five memory accesses to translate a virtual address to physical address and six total memory accesses to actually access the desired data.

Grading Rubric:
+2 for correct reasoning.

5. Multicore, Parallelism, and Synchronization [20 pts] (parts a–d)

A *producer/consumer ring buffer* is a very common data structure used in parallel programs. It is used to pass information between threads. There can be many producer threads and many consumer threads that all share the same producer/consumer ring buffer. The invariant is that a producer can only produce if the buffer is not full, and an item is never overwritten; and a consumer can only consume if the buffer is not empty, and an particular item is consumed only once.

In this problem, we will consider the following code snippets executed in parallel by potentially many `Producer` and `Consumer` threads:

```
Consumer:
   empty = (tail==head);
   if(!empty)
     head++;

Producer:
   full = (tail-head)==n;
   if(!full)
     tail++;
```

These might be compiled into the following assembly code. Assume that variables `head` is at location `0($a0)`, `tail` is at location `0($a1)`, and an extra shared variable at `0($a2)`.

```
Consumer:
    LW $t0, 0($a0)      // A0 : read head
    LW $t1, 0($a1)      // A1 : read tail
    BEQ $t0, $t1, Consumer
    NOP
    ADDIU $t0, $t0, 1
    SW $t0, 0($a0)      // A2 : store head+1

Producer:
    LI $t3, n
    LW $t0, 0($a0)      // B0 : read head
    LW $t1, 0($a1)      // B1 : read tail
```

```
SUB $t2, $t0, $t1,
BEQ $t2, $t3, Producer
NOP
ADDIU $t1, $t1, 1
SW $t1, 0($a1)      // B2 : store tail+1
```

The load and store instructions are marked `A0`, `A1`, `A2`, `B0`, `B1`, `B2`. Assume that the architecture ensures that memory accesses from one thread are never done out of order from the viewpoint of all threads. For example, this means `A0` always comes before `A1`, and `B0` always comes before `B1`.

(a) [6 pts] Let us assume there are *two* `Consumer` threads, i and j and one item in the ring buffer. Show two possible interleavings if each thread executes the `Consumer` code one time with the starting values: `head` = 1, `tail` = 2. For example, the interleaving (A_i0, A_i1, A_i2, A_j0, A_j1) results in outcome (`head = 2`, `tail = 2`), one item being consumed, and A_j2 not being executed because the buffer would be empty.
Show two more possible outcomes for (`head`, `tail`), and for each, state how many items were consumed, and show the corresponding interleaving of memory operations.

Answer:

Only two outcomes are possible with two threads in parallel iterating through the `Consumer` *code once:*
(1) `head`*=2,* `tail`*=2, one item consumed, or*
(2) `head`*=2,* `tail`*=2, two items consumed*

We did accept differnt interleavings that resulted in the above two outcomes
(A_i0, A_j0, A_i1, A_j1, A_i2, A_j2) : (`head` *= 2,* `tail` *= 2), two items consumed, which is an error since only one item was available.*
(A_i0, A_j0, A_j1, A_j2, A_i1, A_i2) : (`head` *= 2,* `tail` *= 2), one item consumed*
etc.

Grading Rubric:
+3 for each possible outcome (must include valid sequence as well as correct `head`*,* `tail`*, and number of items consumed).*

(b) [10 pts] Briefly explain how to use a standard synchronization mechanism to enforce the invariant: Producers can only produce if the buffer is not full and a consumers can only consume if the buffer is not empty, and no single item is consumed more than once, and no item is overwritten by different producers.

Write pseudo-code using the pair of atomic instructions `LL` and `SC` to achieve synchronization for the code.

Answer:

The most straightforward solution is to implement the increments for `head` *(for the* `Consumer`*) and* `tail` *(for the* `Producer`*) atomically using* `LL` *and* `SC` *directly as follows:*

```
Consumer:
    LW $t1, 0($a1)      // A1 : read tail
    LL $t0, 0($a0)      // A0 : read head
    BEQ $t0, $t1, Consumer
    NOP
    ADDIU $t0, $t0, 1
    SC $t0, 0($a0)      // A2 : store head+1
    BEQZ $t0, Consumer
    NOP


Producer:
    LI $t3, n
    LW $t0, 0($a0)      // B0 : read head
    LL $t1, 0($a1)      // B1 : read tail
    SUB $t2, $t0, $t1,
    BEQ $t2, $t3, Producer
    NOP
    ADDIU $t1, $t1, 1
    SC $t1, 0($a1)      // B2 : store tail+1
    BEQZ $t1, Producer
    NOP
```

Note, we swapped the read of `head` *and* `tail` *in the* `Consumer` *so that there would not be any other loads or stores to memory between the* `LL` *and* `SC`*)*

Another way, but much tricker way, to implement concurrency safe `Consumer` and `Producer` was to turn the code for both `Consumer` and `Producer` into critical sections by acquiring the same mutex immediately before each statement, and releasing the mutex immediately after. HOWEVER, you needed to also release the mutex if the condition was not satisfied. That is, cannot be in a while loop inside of a critical section.

The code for locking is:

```
m = 0;
mutex_lock (int *m) {
while (test_and_set(m)) {}
}

int test_and_set (int *m) {
try:
    LI $t0, 1
    LL $t1, locM
    SC $t0, locM
    BEQZ $t0, try
    MOVE $v0, $t1
}

mutex_unlock (int *m) {
    sw $zero, locM
}
```

The other variant is:

```
mutex_lock (int *m) {
test_and_set:
    LI $t0, 1
    LL $t1, locM
    BNEZ $t1, test_and_set
    SC $t0, locM
    BEQZ $t0, test_and_set
}
```

In both the techniques above, the way we then do the updates is:

```
   int m = create_mutex_lock (); // anything reasonable here is fine

   // modified code for Consumer, so that code does
   // not spin while holding lock
Consumer:
   mutex_lock (m);
```

```
    LW $t0, 0($a0)      // A0 : read head
    LW $t1, 0($a1)      // A1 : read tail
    BNE $t0, $t1, STORE
    NOP

    mutex_unlock(m)
    J Consumer
    NOP

  STORE:
    ADDIU $t0, $t0, 1
    SW $t0, 0($a0)      // A2 : store head+1
   mutex_unlock (m);


   // modified code for Producer, so that code does
   // not spin while holding lock
Producer:
   mutex_lock (m);
    LI $t3, n
    LW $t0, 0($a0)      // B0 : read head
    LL $t1, 0($a1)      // B1 : read tail
    SUB $t2, $t0, $t1,
    BNE $t2, $t3, STORE
    NOP

    mutex_unlock(m)
    J Consumer
    NOP

  STORE:
    ADDIU $t1, $t1, 1
    SW $t1, 0($a1)      // B2 : store tail+1
   mutex_unlock (m);
```

Grading Rubric:

Explanation
+4 for correct explanation (key words: atomic increment, mutex, critical section)
-2 if explanation does not match code

Implementation
+3 each for fully correct implementation of the `Producer` and `Consumer`.
Could have correctly implemented using of `LL` / `SC` within code for each thread or using mutex locks. But, using mutex locks could NOT spin while holding lock.

-1 for deadlock
-1 for unnecessary `LL`s

[This page intentionally left empty]

(c) [2 pts] Is there any way to optimize the code if there is only one `Consumer` (i.e. there is only one `Consumer` thread)? Explain, briefly.

Answer:

If there is only one `Consumer` *thread, then there is no need to atomically increment* `head` *since there will not be a race condition to update* `head`*. Further, since the* `Consumer` *only reads* `tail`*, it will never consume an item that has not already been produced, so there is no race condition with the* `Producer`*. As a result, there is no need for use of* `LL` */* `SC`*, or use of mutex locks to implement a critical section, in the* `Consumer`*.*

Grading Rubric:
+2 for fully correct explanation.
Needed to state and justify that there is no need for use of `LL` */* `SC`*, or use of mutex locks to implement a critical section, in the* `Consumer`*.*

(d) [2 pts] Is there any way to optimize the code if there is only `Consumer` and `Producer` (i.e. there is only one `Consumer` thread and only one `Producer` thread)? Explain, briefly.

Answer:

If there is only one `Consumer` *thread and one* `Producer` *thread, then there is no need to atomically increment* `head` *or* `tail` *since there will not be a race condition to update them. Further, since the* `Consumer` *only reads* `tail`*, it will never consume an item that has not already been produced, so there is no race condition with the* `Producer`*. Similarly, since the* `Producer` *only reads* `head`*, it will never attempt to produce an item if the buffer is full. As a result, there is no need for use of* `LL` */* `SC`*, or use of mutex locks to implement a critical section, in the* `Consumer` *or* `Producer`*. In general, there is often no need for synchronization if there is only one writer to a variable.*

Grading Rubric:
+2 for fully correct explanation.
Needed to state and justify that there is no need for any synchronization in the `Producer` *or* `Consumer`*.*

6. Calling Conventions and Assembly Code [25 pts] (parts a–g)

For all parts of this question attempt to follow the calling and register convention described in class and in Chapter 2 and Appendix A of P&H.

(a) [2 pts] Describe the advantages of having both caller- and callee-saved registers.

Answer:

Callee-saved registers are useful because they do not have to be saved to memory unless the function being called actually uses them. Similarly, caller-saved registers have the advantage that the caller can skip saving them to memory in the event that it does not actually need them again after the function call.

Grading Rubric:
+1 for an advantage for each callee- and caller-saved registers

(b) [3 pts] Consider the following C program. How many caller- and callee-saved registers would you use and for which variables?

```
int foo() {
    int a = 0;
    int b = 12;
    int c = 1;

    while(b + c > 0) {
        int e = b + bar(c);
        c = b + e;

        int d = c + baz(b);
        a = d - e;
    }

    return a;
}
```

Answer:

Variables `a` *and* `d` *should be placed in caller-saved registers (for a total of 2) because they don't have to be preserved across a function call. Variables* `b`*,* `c`*, and* `e` *should be placed in callee-saved registers (for a total of 3) because they do have to be preserved.*

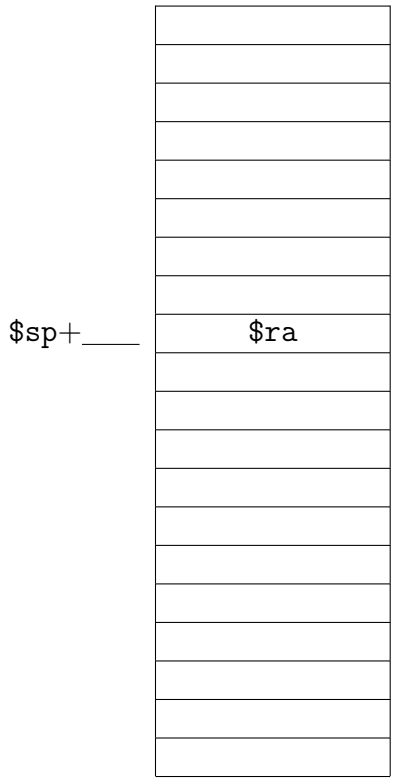
Grading Rubric:
+0.5 for serious attempt +0.5 for each correct declaration of callee- or caller-save registers

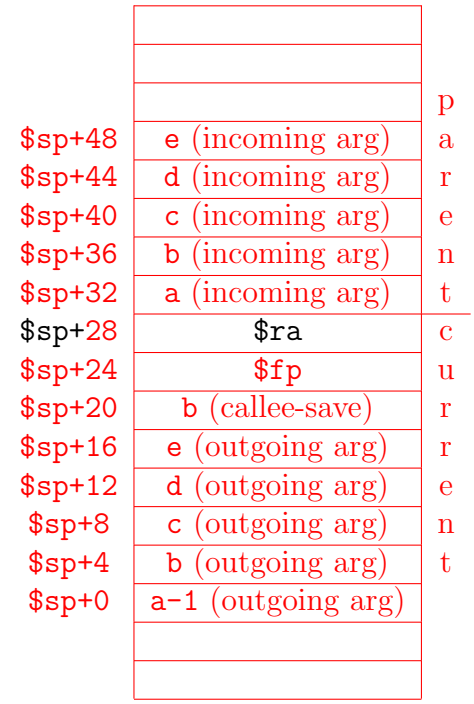
(c) [3 pts] Consider the following C program. How large is the stack frame?

Fill in the stack frame for the function. Show and mark the location where the space for each incoming and outgoing argument, callee-/caller-save, etc is allocated. Also, write the location of each item on the stack with respect to the `$sp`, and identify the current and parent stack frame.

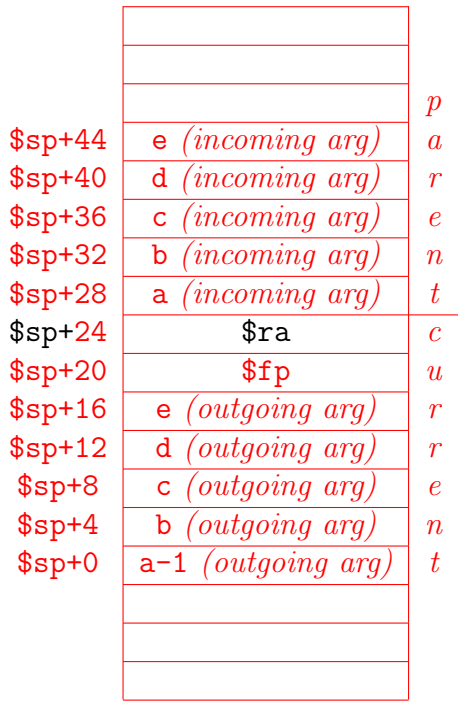
```
int recursive_foo(int a, int b, int c, int d, int e) {
   int g;
   if (a > 0)
      g = recursive_foo(a - 1, b, c, d, e)

   return b + g;
}
```




$sp+48
e (incoming arg)
$sp+44
d (incoming arg)
$sp+40
c (incoming arg)
$sp+36
b (incoming arg)
$sp+32
a (incoming arg)
$sp+28
$ra
$sp+24
$fp
$sp+20
b (callee-save)
$sp+16
e (outgoing arg)
$sp+12
d (outgoing arg)
$sp+8
c (outgoing arg)
$sp+4
b (outgoing arg)
$sp+0
a-1 (outgoing arg)
parent
current

Answer:



The stack needs to be allocated with space for 8 words (32 bytes) if save variable `b` *is a callee-save register; otherwise, stack only needs to be 7 words (28 bytes) if saved* `b` *in the argument space reserved for it on the stack. Both answers were accepted.*

Grading Rubric:
+1 for correct stack offset
+1 for correct current stack frame (outgoing arguments and callee save registers in correct location and outgoing arguments in the correct order)
+1 for correct parent stack frame (incoming args in correct location and correct order)

(d) [3 pts] Starting with the arguments and then local variables for function `recursive_foo()`, state whether the assembly implementation will require a callee- or caller-save register. Further, specify which register to use for each arg and local variable, and if it needs to be spilled to the stack (i.e say "stack" if it needs to be saved to the stack).

arg/variable	caller/callee	register (and stack)
a		
b		
c		
d		
e		
g		

Answer:

arg/variable	caller/callee	register (and stack)
a	*caller*	*$a0*
b	*callee*	*$a1 and stack, or $s0*
c	*caller*	*$a2*
d	*caller*	*$a3*
e	*caller*	*$t0 and stack*
g	*caller*	*$t1*

Grading Rubric:
+0.5 correct each row

All variables should be caller-saved.
`b` *could alternatively be callee-saved and register* `$s0` *or caller-saved and $a1 plus stack.*
`g` *could also be caller-save and register* `$v0`.

Common issues:
- There is not register `$a4` *or* `$a5`*!*
- Many people saved all the arguments into callee-saved registers ($s0-$s4). This was fine, but unnecessary since space is already allocated for all arugment registers on the stack. But, after saving and restoring the callee-saved registers, we looked to if the argument registers were actually copied to the callee-saved registers in the last part of this question.

(e) [4 pts] Write the prologue for `recursive_foo()`. Also, use your register assignment from part 6(d).

`RECURSIVE_FOO:`

Answer:

```
RECURSIVE_FOO:
  ADDIU $sp, $sp, -32 # (== 5x outgoing args, 1x $sxx, $ra, $fp)
  SW $ra, 28($sp)
  SW $fp, 24($sp)
  SW $s0, 20($sp)    # store, then $s0 = b
  ADDIU $fp, $sp, 28
```

or

```
RECURSIVE_FOO:
  ADDIU $sp, $sp, -28 # (== 5x outgoing args, $ra, $fp)
  SW $ra, 24($sp)
  SW $fp, 20($sp)
  ADDIU $fp, $sp, 24
```

Grading rubric:
+1 adjust stack size for new stack frame
+1 push $ra and $fp onto stack
+1 consistency table, push any callee-save register onto stack
+1 adjust $fp based on $sp

(f) [2 pts] Using your answers from parts d and e, write the epilogue for `recursive_foo()`.

`EPILOGUE:`

Answer:

```
EPILOGUE:
  LW $s0, 20($sp)
  LW $fp, 24($sp)
  LW $ra, 28($sp)
  ADDIU $sp, $sp, 32
  JR $ra
  NOP
```

or

```
EPILOGUE:
  LW $fp, 20($sp)
  LW $ra, 24($sp)
  ADDIU $sp, $sp, 28
  JR $ra
  NOP
```

Grading rubric:
+1.5 consistently reversed what was done in with prologue
+0.5 return from function (JR $ra)

-1 for invalid instruction

(g) [8 pts] Now, write the implementation for the body of the `recursive_foo()` function. Use the same registers in the body as your answer in part 6(d).

Write your soln below. Your branches should only have to use labels `RET`, `EPILOGUE`, if necessary. Also, you may use any instruction in your implementation on the MIPS reference sheet including pseudoinstructions.

```
BODY:



IF:  # if (a > 0), call recursive_foo



RET:
```

Answer:

```
BODY:

  MOVE $s0, $a1   # $s0 = b


IF: # if (a > 0), call recursive_foo
  BEQZ $a0, RET
  NOP
  ADDIU $a0, $a0, -1  # a-1
  LW $t0, 48($sp)     # load e from parent stack frame
  SW $t0, 16($sp)     # store e in space for outgoing args
  JAL RECURSIVE_FOO
  NOP


RET:
   ADD $v0, $v0, $s0 # return g + b
```

or

```
BODY:

  SW $a1, 32($sp) # store b in space allocated
                  #   for it in parent stack frame


IF:  # if (a > 0), call recursive_foo
  BEQZ $a0, RET
  NOP
  ADDIU $a0, $a0, -1  # a-1
  LW $t0, 44($sp)     # load e from parent stack frame
  SW $t0, 16($sp)     # store e in space for outgoing args
  JAL recursive_foo
  NOP


RET:
  LW $t0, 32($sp)   # restore b from parent stack frame
  ADD $v0, $v0, $t0 # return g + b
```

Grading rubric:
+1 for loading `e` from parent stack frame
+1 for storing `e` in space for outgoing args
+2 call `recursive_foo` correctly (args/`JAL`)

+2 consistency with previous parts of the problem; e.g. whether you used callee-saved registers for arguements like saving/restoring `b` to/from the stack frame

+1 delay slots (use or insert `NOPs`)
+1 answer in $v0

-1 Invalid instruction
-2 (each) faulty program logic (i.e. deviate from C code)

[recursive_foo() function is repeated for reference]

```
int recursive_foo(int a, int b, int c, int d, int e) {
   int g;
   if (a > 0)
      g = recursive_foo(a - 1, b, c, d, e)

   return b + g;
}
```