# 2018 Exam - Computer Architecture
# CS3021

Two Hour Exam

Answer 3 out of 4 Questions

20 marks per question

40 mins per question

# Question 1

*i) Briefly describe IA32 and x64 procedure calling conventions. Make sure that you describe the register set, draw a diagram of a typical procedure stack frame and list the volatile registers. In the case of x64 explain the use of shadow space. [6 marks]*

IA32

When calling a function in IA32 all parameters must be pushed onto the stack from right to left. So if calling *min(a,b,c)* you must first push a, then b, then c. Once parameters have been pushed you must then call and enter the function.

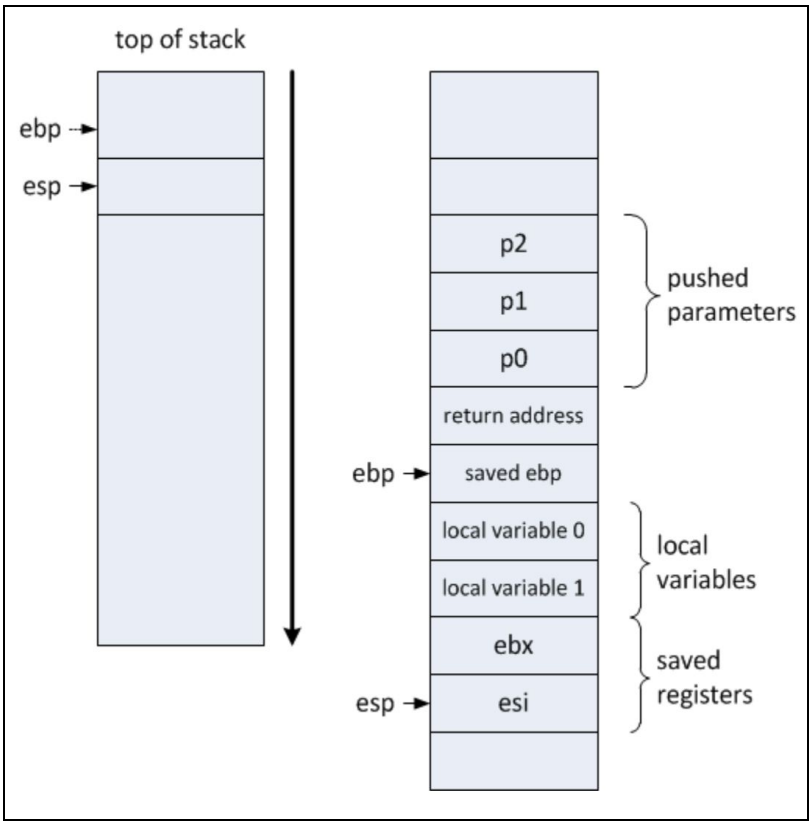
1. Push parameters to stack (right to left)
2. Call and enter function

Within the function you must initially push the frame pointer to the stack to preserve it for when exiting. If there are any local variables used, space must be allocated for these upon the stack initially. Also, if any non volatile registers are used they must also be pushed to the stack and preserved.

3. Preserve frame pointer (**push ebp**)
4. Update frame pointer with stack pointer (**mov ebp, esp**)
5. Allocate space for local variables (**sub esp, 8** -> space for two local variables)
6. Preserve non-volatile registers if used (**push ebx**)

From here the function body can proceed and fulfill its desired functionality. Before exiting the function must perform the following:

7. Restore non-volatile registers saved (**pop ebx**)
8. De-allocate space for local variables (**add esp, 8**)
9. Restore esp (**mov esp, ebp**)
10. Restore frame pointer (**pop ebp**)
11. Return (**ret 0**)




Parameters are pushed right to left onto the stack.

p0 @ ebp + 8

p1 @ ebp + 12

p2 @ ebp + 16

Local variables are accessed using a negative offset.

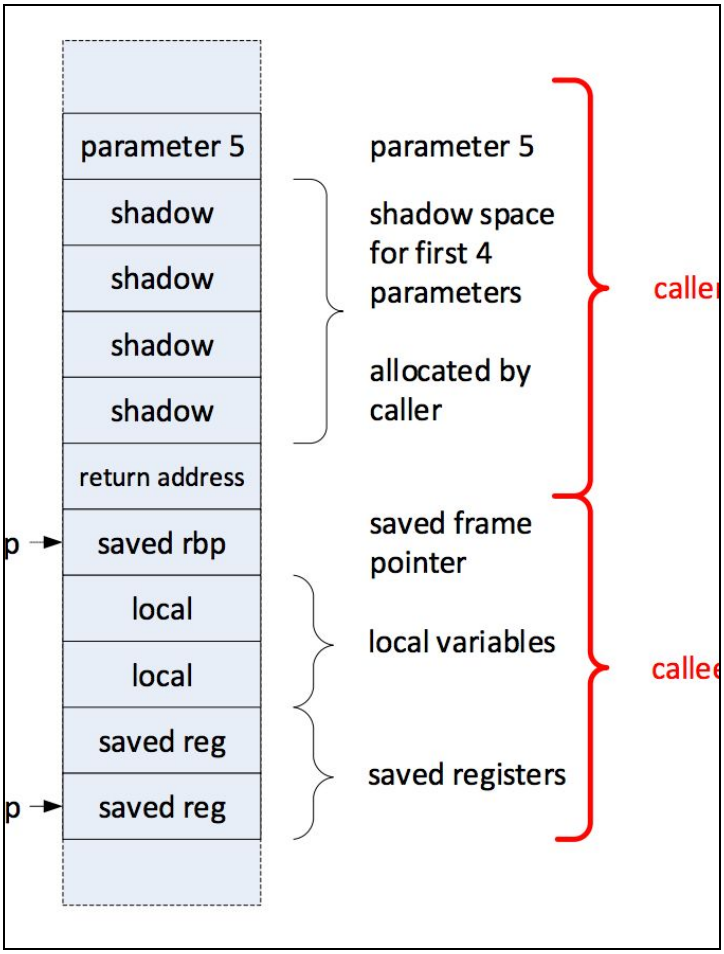
local0 @ ebp - 4

local1 @ ebp - 8

X64

X64 function calling is quite similar to that of IA32 just with an extended register set and what's known as *shadow space*. Shadow space is 32 bytes of pre-allocated space that every function caller must allocate before calling a function and de-allocate following a return. This way within every function there is 4 x 8 bytes available for parameters. Any additional parameters must be pushed onto the stack. Parameters, like in IA32 are passed from right to left.

1. Pass parameters to 1-4 to *rcx, rdx, r8 and r9* (**mov rcx, rax**)
2. Allocate shadow space (**sub esp, 32**)
3. Call function (**call min**)
4. De-allocate shadow space (**add esp, 32**)
5. Result found in rax




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

all volatile registers

Parameters are pushed right to left onto the stack.

p0 @ rcx

p1 @ rdx

p3 @ r8

p4 @ r9

*ii) What is the main advantage of the x64 procedure calling convention compared with the IA32 procedure calling convention? [2 marks]*

The main advantage of x64 procedure calling vs IA32 is the implementation of shadow space. This allows for 4 registers to be pre-allocated for use with parameters or any other desired usage for every given function. This removes the need for unique stack and frame pointer maintenance and handling throughout functions.

It implements a generic protocol/procedure that every function call must follow and allows for programmers to focus on writing their code and not worrying about stack and frame pointers being handled incorrectly.

*iii) Convert the following code segment into IA32 and x64 assembly.*

```
INT g = 4;

INT max(INT a, INT b, INT c) {
    INT v = a;
    if (b > v)
        v = b;
    if (c > v)
        v = c
    return v;
}

INT p(INT i, INT j, INT, k, INT l) {
    return max(max(g, i, j), k, l);
}
```

IA32

```
; function to calculate max(a, b, c)
;             a -> [ebp+8]
;             b -> [ebp+12]
;             c -> [ebp+16]
;
; returns result in eax

public    max                   ; make sure function name is exported

min:      push  ebp             ; push frame pointer
          mov   ebp, esp        ; update ebp

          mov   eax, [ebp+8]    ; v = a
          mov   ecx, [ebp+12]   ; ecx = b
          cmp   ecx, eax        ; if (b > v)
          jle   max_1           ;
          mov   eax, ecx        ; v = b

max_1:    mov   ecx, [ebp+16]   ; ecx = c
          cmp   ecx, eax        ; if (c > v)
          jle   max_2           ;
          mov   eax, ecx        ; v = c
max_2:    mov   esp, ebp        ; restore esp
          pop   ebp             ; restore ebp
          ret   0               ; return
```

```
; function to calculate p(i, j, k, l)
;              i -> [ebp+8]
;              j -> [ebp+12]
;              k -> [ebp+16]
;              l -> [ebp+20]
;
; returns max( max(g, i, j), k, l)

public    p                    ; make sure function name is exported

min:      push  ebp            ; push frame pointer
          mov   ebp, esp       ; update ebp

          push  [ebp+12]       ; push j for max(g, i, j)
          push  [ebp+8]        ; push i for max(g, i, j)
          push  g              ; push g for max(g, i, l)

          call max             ; eax = max(g, i, j)

          push [ebp+20]        ; push l for max(eax, k, l)
          push [ebp+16]        ; push k for max(eax, k, l)
          push eax             ; push eax for max(eax, k, l)

          call max             ; max( max(g, i, j), k, l)

          mov esp, ebp         ; restore stack pointer
          pop ebp              ; restore frame pointer
          ret 0                ; return 0
```

x64

```
; function to calculate max(a, b, c)
;              a -> rcx
;              b -> rdx
;              c -> r8
;
; returns result in eax

public    max                  ; make sure function name is exported

min:      mov   rax, rcx       ; v = a
          cmp   rdx, rax       ; if (b > v)
          jle   max_1          ;
          mov   rax, rdx       ; v = b

max_1:    cmp   r8, rax        ; if (c > v)
          jle   max_2          ;
          mov   rax, r8        ; v = c

max_2:    ret   0              ; return
```

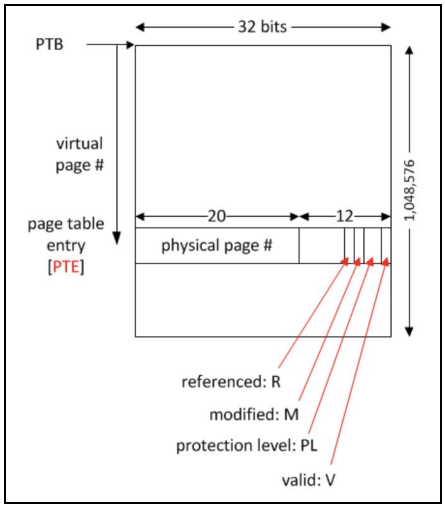
```
; function to calculate p(i, j, k, l)
;              i -> rcx
;              j -> rdx
;              k -> r8
;              l -> r9
;
; returns max( max(g, i, j), k, l)

public    p                    ; make sure function name is exported

p:        push r8               ; preserve r8 (k)
          mov r8, rdx           ; r8 = j
          mov rdx, rcx          ; rdx = i
          mov rcx, g            ; rcx = g

          add esp, 32           ; allocate shadow space
          call max              ; rax = max (g, i, j)
          sub esp, 32           ; de-allocate shadow space

          mov r8, r9            ; r8 = l
          pop rdx               ; rdx = k
          mov rcx, rax          ; rcx = max(g,i,j)

          add esp, 32           ; allocate shadow space
          call max              ; rax = max(max(g, i, j), k, l)
          sub esp, 32           ; de-allocate shadow space

          ret 0                 ; return
```

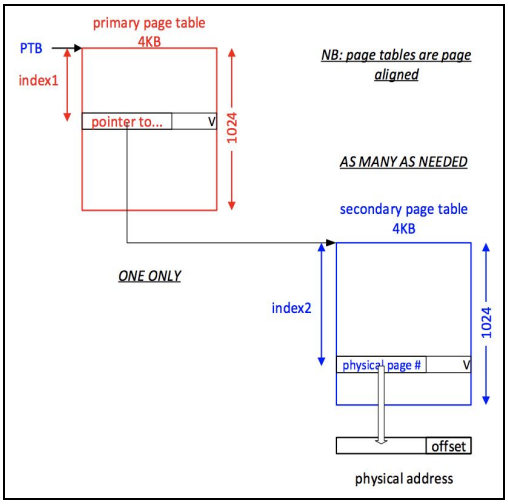
# Question 2

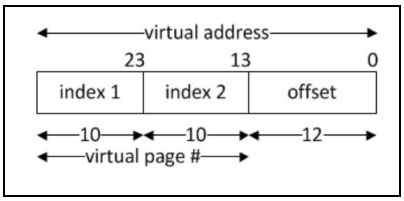
*i) How are virtual addresses converted into physical addresses by a two level page table? Assume a page table structure as per an IA32 based CPU. Draw diagrams to illustrate your answer.*

The Memory Management Unit (MMU) converts virtual addresses generated by the CPU into physical addresses in memory using page tables. Address space is divided into fixed sized pages. Each process runs in its own 4GB virtual address space which has pages mapped by the MMU to memory.

Virtual page numbers are converted to physical page numbers using a page-table lookup. The virtual page number is used as an index into a page table stored in physical memory. The page table base register (PTB) contains the physical address of the page table of the current process. 4MB physical memory needed to hold the page table of every process




Using the virtual address and the primary page (primarytable[index1]) table we can find the address of the secondary page table and from this can get the physical address (secondarytable[index2]) of the data in memory.

*ii) What is the advantage of using a two level page table compared with a single level page table? Comment on the amount of memory needed for a small process and a maximum sized process using a single and two level page table structure.*

To reduce the size of the page table structure an n-level look up table can be used. This means that the larger the process, the more memory is needed for its tables. However, the smaller the process the less memory that is needed.

Each process has one 4MB primary page table and multiple 4KB secondary, tertiary... page tables. Secondary page tables are created on demand as a process' size exceeds that of the capability of a single secondary table and needs more address spaces. This way, small processes never over-use memory and memory is utilized efficiently for both small processes and maximum sized processes.

A process in a 32 bit system can theoretically use each of the possible 1 million 20 bit page locations in its table if necessary. A single 32-bit page table therefore would need 4MB of RAM (32bit = 4 bytes, 4 bytes * 1million rows = 4MB). A two level page table is far more memory usage efficient as the virtual page address is split into two addresses (index1 - primary table index points to secondary page table AND index2 - secondary table index points to address in memory). This means that each primary and secondary table is now 1024 lines long. The same capabilities apply as the 1024 primary table addresses can map to 1024 secondary table addresses (1024^2 = 1 million). More secondary tables after the first one only need to be created as they are required thus saving space in RAM.

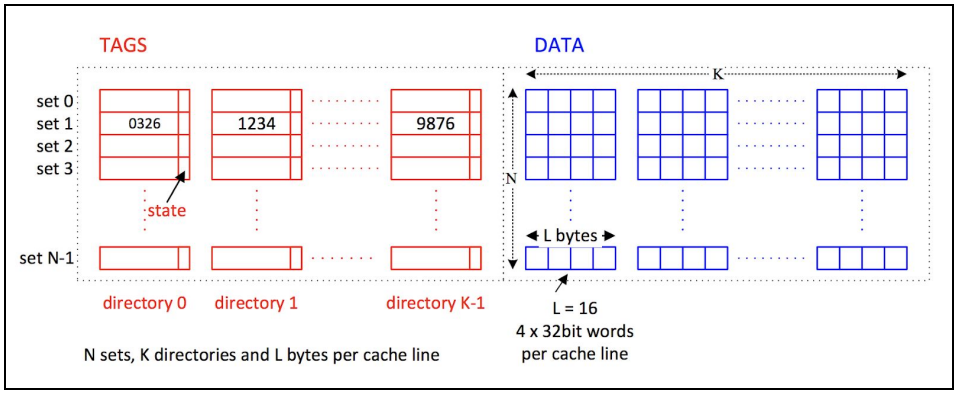
| | min process (i) | max process (ii) |
|---|---|---|
| 1 level page table | 4MB | 4MB |
| 2 level page table | 4KB + 4KB<br>(1 primary level table always present, smallest process will need at least one secondary) | 4KB + 4MB<br>(1 primary level table + 1024 secondary level tables) |

*ii) Draw a diagram showing the structure of an IA32 2 level kernel page table which maps the first 16MB of the kernels virtual address space starting at address 0x00000000 onto 8MB of memory located at physical address 0x10000000 and 8MB of memory located at physical address 0x20000000.*

**4 x Primary Tables** → 4 x [ 1024 Secondary Table Mappings x 4KB each = 4MB each] → 16MB

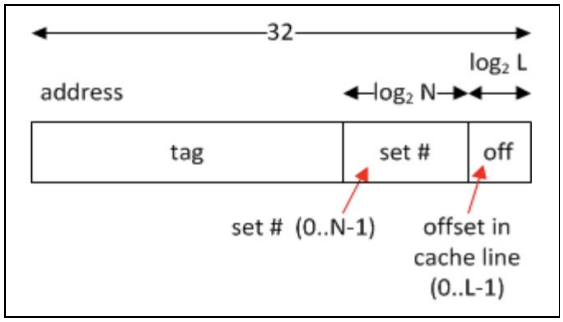
# Question 3

*i) With the aid of a diagram explain how a cache organisation can be characterised by three constants, LKN. Explain in detail, how a data item is accessed in an LKN cache. Why does a cache normally reduce the effective memory access time? [6 marks]*




- N sets of cache lines/tages
- K directories
- L bytes per cache line (L=16 → 4 x 32bit words per cache line)

When referencing/searching a K-Way cache, each cache address is mapped to a particular set (0-N).




[tag] [set #] [offset]

If an address maps to set 1, the set 1 tags of all K directories are compared with the incoming address tag simultaneously (fully associative).

If a match is found (hit), corresponding data returned offset within cache line. The K data lines in the set are also fully associative and accessed concurrently. If a match is not found (miss), read data from memory, place in cache line within set and update corresponding

cache tag (choice of K positions). This replacement depends on the cache line replacement strategy - Least Recently Used (LRU)

*ii) Compute the number of hits and misses if the following list of hexadecimal addresses is applied to (1) a 64 byte direct mapped cache with 16 bytes per line and (2) a 64 byte fully associative cache with 16 bytes per line.*

**(1)** L x K x N = 64

L = 16
K = 1
N = 4

**Can ignore first 00 since all the same and the last 0 since don't care about offset**

| HEX | BINARY | TAG | SET | OFFSET | HIT/MISS | CACHED AT |
|---|---|---|---|---|---|---|
| 0x0000 | 0000~~0000~~ | 00 | 00 | - | MISS | N = 0 |
| 0x0010 | 0001~~0000~~ | 00 | 01 | - | MISS | N = 1 |
| 0x0020 | 0010~~0000~~ | 00 | 10 | - | MISS | N = 2 |
| 0x0030 | 0011~~0000~~ | 00 | 11 | - | MISS | N = 3 |
| 0x0034 | 0011~~0100~~ | 00 | 11 | - | HIT | N = 3 |
| 0x0020 | 0010~~0000~~ | 00 | 10 | - | HIT | N = 2 |
| 0x0010 | 0001~~0000~~ | 00 | 01 | - | HIT | N = 1 |
| 0x000C | 0000~~1100~~ | 00 | 00 | - | HIT | N = 0 |
| 0x0050 | 0101~~0000~~ | 01 | 01 | - | MISS | N = 1 |
| 0x0040 | 0100~~0000~~ | 01 | 00 | - | MISS | N = 0 |
| 0x002C | 0010~~1100~~ | 00 | 10 | - | HIT | N = 2 |
| 0x0008 | 0000~~1000~~ | 00 | 00 | - | MISS | N = 0 |
| 0x0030 | 0011~~0000~~ | 00 | 11 | - | HIT | N = 3 |
| 0x0020 | 0010~~0000~~ | 00 | 10 | - | HIT | N = 2 |
| 0x0010 | 0001~~0000~~ | 00 | 01 | - | MISS | N = 1 |
| 0x0000 | 0000~~0000~~ | 00 | 00 | - | HIT | |

**(2)** L x K x N = 64

L = 16
K = 4
N = 1
**Since N = 1, set always 0**

| HEX | BINARY | TAG | SET | OFFSET | HIT/MISS | REPLACING LRU |
|---|---|---|---|---|---|---|
| 0x0000 | 0000~~0000~~ | 0000 | 0 | - | MISS | |
| 0x0010 | 0001~~0000~~ | 0001 | 0 | - | MISS | |
| 0x0020 | 0010~~0000~~ | 0010 | 0 | - | MISS | |
| 0x0030 | 0011~~0000~~ | 0011 | 0 | - | MISS | |
| 0x0034 | 0011~~0100~~ | 0011 | 0 | - | HIT | |
| 0x0020 | 0010~~0000~~ | 0010 | 0 | - | HIT | |
| 0x0010 | 0001~~0000~~ | 0001 | 0 | - | HIT | |
| 0x000C | 0000~~1100~~ | 0000 | 0 | - | HIT | |
| 0x0050 | 0101~~0000~~ | 0101 | 0 | - | MISS | 0011 |
| 0x0040 | 0100~~0000~~ | 0100 | 0 | - | MISS | 0010 |
| 0x002C | 0010~~1100~~ | 0010 | 0 | - | MISS | 0001 |
| 0x0008 | 0000~~1000~~ | 0000 | 0 | - | HIT | |
| 0x0030 | 0011~~0000~~ | 0011 | 0 | - | MISS | 0101 |
| 0x0020 | 0010~~0000~~ | 0010 | 0 | - | HIT | |
| 0x0010 | 0001~~0000~~ | 0001 | 0 | - | MISS | 0100 |
| 0x0000 | 0000~~0000~~ | 0000 | 0 | - | HIT | |

LRU = ~~0000~~ / ~~0001~~ / ~~0010~~ / ~~0011~~ / ~~0010~~ / ~~0001~~ / ~~0000~~ / ~~0101~~ / ~~0100~~ /~~0010~~ / 0000 / 0011 / 0010 / 0001

*iii) Would you expect an associative cache of size N and line size L to always have a higher hit rate than a direct mapped cache of size N and line size L. Explain the reasoning behind your answer.*

Consider two caches:

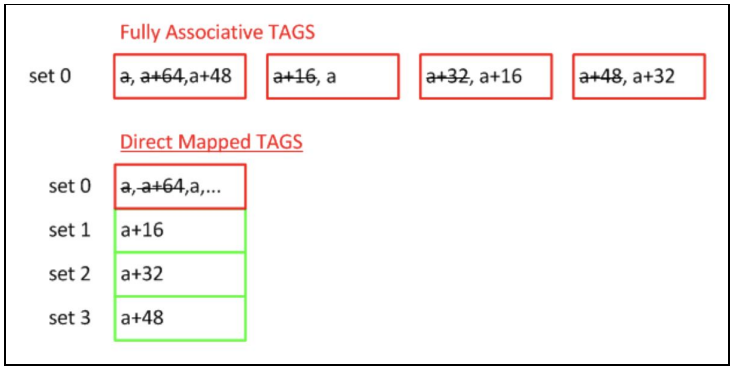
- K = 4, N = 1, L = 16 [64 byte fully associative]
- K = 1, N = 4, L = 16 [64 byte direct mapped]

L * K * N all equal

Consider the following address sequences repeatedly querying cache:

[a], [a+16], [a+32], [a+48], [a+64] → **5 addresses**

Our caches can contain 4 addresses, sequence comprises of 4




Fully Associative: Only 4 addresses can fit in the 4-way cache so, due to the LRU replacement policy, every access will be a miss.

Direct Mapped: Since ONLY addresses [a] and [a+64] will conflict each other as they map to the same set, there will be 2 misses and 3 hits per cycle of 5 addresses.

# Question 4

*i) What is the cache coherency problem? Briefly explain the states and operation of the Write Once cache coherency protocol.*

The cache coherency problem is an issue regarding cache's and physical memory where a CPU might make changes to a value in cache. This change must be propagated to both physical memory and any other cache's that have access to this same shared address/value.

**Write-Once Protocol**

- Uses write-back caches to reduce bus traffic.
- Each cache line can be in one of 4 states; **Valid, Invalid, Reserved, Dirty**
- **Invalid:** Wrong copy, fetch from memory.
- **Valid:** Cache line valid, present in one or more caches
- **Reserved:** Cache line only in local cache, send copy to memory
- **Dirty:** Cache line only in local cache, memory has not been updated yet
- **Valid → Reserved** when written to for first time (write-back to memory).
- **Reserved → Dirty** when written to multiple times, enters a write-back cycle and marked as dirty since now the only valid copy in the system.
- Caches monitor bus traffic, if read from a RESERVED copy, mark as valid
- If a read observed from DIRTY copy, cut read operation and supply local copy, also now write data to memory and mark as valid.

*ii) Describe the changes that occur in a three CPU multiprocessor system where each CPU has its own cache.*

t = 0 → CPU 0 attempts to read a0. This will initially be empty and must be fetched from memory and loaded into CPU 0 set 0. It is initially marked as valid.

t = 1 → CPU 0 attempts to read a0. Since a0 is already in the cache and is marked as valid the correct value is read from the CPU without needing to access memory.

t = 2 → CPU 1 attempts to read a0. This will be initially empty and must be fetched from memory and loaded into CPU 1 set 0. It is initially marked as valid.

t = 3 → CPU 1 attempts to write to a0. This change will be propagated to memory over the bus and within CPU 1 set0 will be marked as Reserved. Since a0 has now changed in memory, CPU 0's value of a0 is now marked as invalid

t = 4 → CPU 2 attempts to read a0. This will initially be empty and must be fetched from memory and loaded into CPU 2 set 0. It is initially marked as valid. **CPU 1's value of a0 will now also be marked as valid since it is shared.**

t = 5 → CPU 2 attempts to write to a0. This change will be propagated to memory over the bus and within CPU 2 set0 will be marked as Reserved. Since a0 has now changed in memory, CPU 0 and CPU 1's value of a0 is now marked as invalid.

t = 6 → CPU 0 attempts to write to a0. Since a0 is invalid in this cache the value of a0 must be read from memory. It is fetched from memory into CPU 0 and marked as valid. CPU 2's value of a0 is now also marked as valid since it is now shared. CPU 0 then proceeds to write to a0. This change is propagated back to memory and a0 is marked as invalid in both CPU 1 and 2. It is now marked as Reserved in CPU 0.

t = 7 → CPU 0 now attempts to read a0. Since a0 is reserved in this cache the value of a0 is read directly from the cache and no traffic is passed along the bus.

t = 8 → CPU 1 attempts to read a0. Since a0 is invalid in this cache the value of a0 must be read from memory. It is fetched from memory into CPU 1 and marked as valid. CPU 0's value of a0 is now also marked as valid since it is now shared.

t = 9 → CPU 0 attempts to write to a0. Since a0 is valid, no data is fetched from memory and the write is performed immediately. This change is propagated to memory and the status of a0 in CPU 0 is set to reserved. CPU 1's value of a0 is now marked as invalid.

t = 10 → CPU 1 attempts to write to a0. Since a0 is invalid, the up to date value is first fetched from memory. When this is fetched it is marked as valid and so is CPU 0's copy of a0. After this the write is performed and the change is propagated back to memory. CPU 1's value of a0 is marked as reserved and CPU 0's value is marked as invalid.

t = 11 → CPU 1 again attempts to write to a0. Since a0 is reserved the write is performed in place and no data is fetched or transferred along the bus. The value of a0 is now marked as dirty in CPU 1 as it is the only cache with a valid copy of a0. If any other CPU attempted to then read a0, CPU 1 would have to interrupt the read and pass the new value of a0 to both this CPU and memory.

*i) What is the cache coherency problem? Briefly explain the states and operation of the MESI cache coherency protocol.*

The cache coherency problem is an issue regarding cache's and physical memory where a CPU might make changes to a value in cache. This change must be propagated to both physical memory and any other cache's that have access to this same shared address/value.

**MESI Protocol (Vivio)**

- Very similar to write-once however, like Firefly has a SHARED bus signal
- This way cache line can enter cache and be placed directly in shared/exclusive state
- **M**odified → Dirty
- **E**xclusive → Reserved
- **S**hared → Valid
- **I**nvalid → Invalid

What is the difference between MESI and Write-Once? Cache line may enter cache and be placed directly in the Exclusive state. Write-once and write-through cycles no longer necessary if cache line is exclusive.

*ii) Describe the changes that occur in a three CPU multiprocessor system where each CPU has its own cache.*

t = 0 → CPU 0 attempts to read a0. Since it is not present in its register set it must fetch it from memory. It is then placed in the exclusive state.

t = 1 → CPU 0 attempts to read a0 again. Since it is already present in its register set and is exclusive it is read directly from here and not fetched from memory.

t = 2 → CPU 1 attempts to read a0. Since it is not present in its register set it must be fetched from memory. CPU 0 sees this read and pings the shared bus and now a0 is marked as SHARED in both CPU 1 and CPU 0.

t = 3 → CPU 1 attempts to write to a0. Since it is already present no data is fetched from memory. A0 is updated locally and since the value is shared this change is propagated to memory and CPU 0's value of a0 is marked as invalid and CPU 1's value of a0 is marked as exclusive.

t = 4 → CPU 2 attempts to read a0. Since it is not in its data set's it is fetched from memory and stored in its cache. The value of a0 is then marked as SHARED in both CPU 1 and CPU 2.

t = 5 → CPU 2 attempts to write to a0. Since it is already present no data is fetched from memory. A0 is updated locally and since the value is shared this change is propagated to memory and CPU 1's value of a0 is marked as invalid and CPU 2's value of a0 is marked as exclusive.

t = 6 → CPU 0 attempts to write a0. Since its value is marked as INVALID it must first be fetched from memory. It is then written to in CPU 0's cache and updated in memory. The value of a0 is marked as EXCLUSIVE in CPU 0 and CPU 1 and 2 are INVALID.

t = 7 → CPU 0 attempts to read a0. Since its value is EXCLUSIVE the read is performed directly from the cache itself and no data is fetched from memory.

t = 8 → CPU 1 attempts to read a0. Since its value is INVALID the read is performed from memory. CPU 0 also sees this read occurring and changes its state of a0 to SHARED.

t = 9 → CPU 1 attempts to write to a0. This write is performed and the change propagated to memory. It is then marked as EXCLUSIVE and a0's flag within CPU 1 is changed to INVALID.

t = 10 → CPU 1 attempts to write to a0 again. Since exclusive, this write is only applied locally and no data is transferred to memory. The flag is also changed to MODIFIED.

t = 11 → CPU 2 attempts to read a0. Since INVALID it attempts to fetch from memory. CPU 1 see's this read occurring on a MODIFIED value of his and intervenes. CPU 1 supplies the data to CPU 2 and also propagates the change to memory. They are then both marked as SHARED.