2 types of registers: Memory Data Registers eg eax, ebx, ecx, edx

And memory address registers eg program counter/"instruction pointer" in intel. Stack pointer, base pointer.  
Might ask this in a test

And recall cache memory used between registers and main memory

Memory Address Register, Memory Data Registers and Memory 
a byte 
2n-1 
Address 
000 
address 
line 
it n 
Data 
memory data register 
individual 
memory 
cells 

MAR-MDR Example 
msb 
Isb 
1100012 = 49 
49 
10 
Memory data register 
Active 
line 

Visual analogy of a memory line. 
isb 
110001 
All cells 
49 
2 = 4910 
10101101 
Memory data register 

To help with memory access, we have some processes built into our computer architecture to protect the memory, manage memory etc.   
Ensures processes can't interfere with each other

Base/Limit registers are different types of registers

Base register points to the base address of where we load our process in

Limit register tells limit of how much memory our process has, eg 10kb.

In this example process 3 can start at 300040+120900=420940

Can't jump outside of our limit register

Base and Limit Registers 
• A pair of base and limit registers define the logical address space 
• CPU must check every memory access generated in user mode to be sure it is between base and limit for that 
user 
operating 
system 
256000 
process 
300040 
base 
process 
120900 
420940 
limit 
1024000 

To access memory, has to be above base and below base+limit. Else an exception interrupt/"trap" will happen, a "general protection fault" in windows

Recall stack pointer, program counter etc will be somewhere in the middle of these 2 values.

Hardware Address Protection 
trap to operating system 
monitor—ado resins error 

Looking at the memo for prac 6:

Recall assembly has labels eg jmp thisIsALabel. They're just symbols representing memory locations. Symbolic memory location.

Also used in higher level representation, eg {} for functions "symbolise" the beginning and end of the memory of the function. In C++ can use "GOTO" and labels as well.

But when you create a symbol, how does it get linked to an actual memory address?

Method 1 is "programming", Back in the old days you had to set memory locations manually in assembly, couldn't just use symbols.

Method 2 is "Linking", the linker comes and does this, links symbols to memory.

From QnA: Import address table used to link memory addresses of functions.

Method 3 is "runtime". Eg [ebp+8], it reads the value that's in ebp and adds 8 and accesses it like memory address.

So binding can happen at compile time (1), load time (2) and execution time (3)

Address Binding 
• Programs on disk, ready to be brought into memory to execute form an input queue 
• Without support, must be loaded into address 0000 
• Inconvenient to have first user process physical address always at 0000 
• How can it not be? 
• Further, addresses represented in different ways at different stages of a program's life 
• Source code addresses usually symbolic 
• Compiled code addresses bind to relocatable addresses 
• i.e. "14 bytes from beginning of this module" 
• Linker or loader will bind relocatable addresses to absolute addresses 
• i.e. 74014 
• Each binding maps one address space to another 

Binding of Instructions and Data to Memory 
• Address binding of instructions and data to memory addresses can happen at three different stages 
• Compile time: If memory location known a priori, absolute code can be generated; must recompile code if starting 
location changes 
• Load time: Must generate relocatable code if memory location is not known at compile time 
• Execution time: Binding delayed until run time if the process can be moved during its execution from one memory 
segment to another 
• Need hardware support for address maps (e.g., base and limit registers) 

So when compiled, it makes an object module in an .exe or .dll

Each .exe hass a "loader program" that says "this is the memory address I need, this is the linker modules that were linked in, these are starting registers and offset registers etc"

Once the loader has loaded, the process is in memory

Multistep Processing of a User Program 
dynarfiC 
linkirN 
object 
memory 
tiff* (run 

Addresses we see in the assembly debugger are "logical addresses" aka "virtual addresses"

So 2 different processes can have the same "logical addresses" but different physical addresses, base address in a different place ("not really" though, he'll clarify later)   
This is because we need isolation. The programmer has no idea where the physical addresses the process is using.

The CPU physically generates these addresses.

Logical vs. Physical Address Space 
• The concept of a logical address space that is bound to a separate physical address space is central to 
proper memory management 
• Logical address — generated by the CPU; also referred to as virtual address 
• Physical address — address seen by the memory unit 
• Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical 
(virtual) and physical addresses differ in execution-time address-binding scheme 
• Logical address space is the set of all logical addresses generated by a program 
• Physical address space is the set of all physical addresses generated by a program 

Memory management unit manages memory allocation.   
Has a bunch of registers and some of its own memory. Sometimes can allocate more memory to itself to find out where other memory is.

Registers are relocation registers, used to convert from logical to physical address

Memory-Management Unit (MMU) 
• Hardware device that at run time maps logical(virtual) to physical address 
• Many methods possible, covered in the rest of this chapter 
• To start, consider simple scheme where the value in the relocation register is added to every address 
generated by a user process at the time it is sent to memory 
• Base register now called relocation register 
• MS-DOS on Intel 80x86 used 4 relocation registers 
• The user program deals with logical addresses; it never sees the real physical addresses 
• Execution-time binding occurs when reference is made to location in memory 
• Logical address bound to physical addresses 

Static linking - all the library code gets put into 1 .exe file when compiling.  
So then that .exe can run on a system that doesn't have eg. the base c libraries installed, eg different operating systems.

Vs Dynamic Linking (dynamic linked libraries dlls)

We can fetch memory for the libraries we need and load it into memory/RAM.

So we save memory because a lot of libraries have functions we rarely use, why must they be loaded into the exe?

Very good for shared libaries.

Dynamic relocation using a relocation register 
• Routine is not loaded until it is called 
• Better memory-space utilization; unused routine is 
never loaded 
• All routines kept on disk in relocatable load format 
• Useful when large amounts of code are needed to 
handle infrequently occurring cases 
• No special support from the operating system is 
required 
• Implemented through program design 
• OS can help by providing libraries to implement dynamic loading 
registe r 
14000 
bgical 
CPU 

Dynamic Linking 
• Static linking — system libraries and program code combined by the loader into the binary program image 
• Dynamic linking —linking postponed until execution time 
• Small piece of code, stub, used to locate the appropriate memory-resident library routine 
• Stub replaces itself with the address of the routine, and executes the routine 
• Operating system checks if routine is in processes' memory address 
• If not in address space, add to address space 
• Dynamic linking is particularly useful for libraries 
• System also known as shared libraries 
• Consider applicability to patching system libraries 
• Versioning may be needed 

Don't need to use up all the memory to get a certain code, just need to get it when we need it

3 techniques for memory management:

* Swapping (he sometimes called it paging):
  + A process must be loaded into memory to run. Reuqirements for memory can exceed memory we have. So "backing store", a disk fast enough to accomodate copies of memory images for all users, must provide direct access. So we can "roll out' a memory into the backing store, "rolling/swapping in" brings it back in
  + Equivalent to like, storing a class of students in a building in the CBD and bringing them in for the lecture, vs running the process is taking from memory (the rows of desks) onto stage (the cpu).   
    So it's slow by comparison to the other processing.
  + In some cases you only need to transfer in, not transfer out. Eg dlls, functions with executable code but no memory, will talk more about this later.
  + This swapping still happens today, but only really when your memory is overutilised. Disk thrashing = not enough memory for the "working set" of our process, so we keep paging processes in and out, 90% that and 10% actual processing. Solution to thrashing is to make a process/processes go to sleep.
  + Medium term scheduler looks at processes in the backend store and tries to bring things into memory.
  + Swapping not really done so much on mobile. If so, done on flash memory which is a bit faster
* Segmentation: breaking up code into segments of memory to fit more efficiently
* Paging

All are used but for different purposes

Swapping 
• A process can be swapped temporarily out of memory to a backing store, and then brought back into memory 
for continued execution 
• Total physical memory space of processes can exceed physical memory 
• Backing store — fast disk large enough to accommodate copies of all memory images for all users; must 
provide direct access to these memory images 
• Roll out, roll in — swapping variant used for priority-based scheduling algorithms; lower-priority process is 
swapped out so higher-priority process can be loaded and executed 
• Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory 
swapped 
• System maintains a ready queue of ready-to-run processes which have memory images on disk 

Swapping (Cont.) 
• Does the swapped out process need to swap back in to same physical addresses? 
• Depends on address binding method 
• Plus consider pending I/O to / from process memory space 
• Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows) 
• Swapping normally disabled 
• Started if more than threshold amount of memory allocated 
• Disabled again once memory demand reduced below threshold 

Schematic View of Swapping 
operating 
system 
@swap out 
2 swap in 
user 
space 
man memory 
process 
process P? 
backing store 

Context switch is very slow

Context Switch Time including Swapping 
• If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process 
• Context switch time can then be very high 
• 100MB process swapping to hard disk with transfer rate of 50MB/sec 
• Swap out time of 2000 ms 
• Plus swap in of same sized process 
• Total context switch swapping component time of 4000ms (4 seconds) 
• Can reduce if reduce size of memory swapped — by knowing how much memory really being used 
System calls to inform OS of memory use via request_memory and release _ memory ( ) 

Should we swap out while waiting for IO? Not great cuz then when the IO returns we'd need to wait for swapping to get process back into memory

Context Switch Time and Swapping (Cont.) 
• Other constraints as well on swapping 
• Pending I/O — can't swap out as I/O would occur to wrong process 
• Or always transfer I/O to kernel space, then to I/O device 
• Known as double buffering, adds overhead 
• Standard swapping not used in modern operating systems 
• But modified version common 
• Swap only when free memory extremely low 

Swapping on Mobile Systems 
• Not typically supported 
• Flash memory based 
• Small amount of space 
• Limited number of write cycles 
• Poor throughput between flash memory and CPIJ on mobile platform 
• Instead use other methods to free memory if low 
• iOS asks apps to voluntarily relinquish allocated memory 
• Read-only data thrown out and reloaded from flash if needed 
• Failure to free can result in termination 
• Android terrninates apps if low free memory, but first writes application state to flash for fast restart 
• Both OSes support paging as discussed below 

Contiguous allocation: Need to put the WHOLE process into memory space.   
OS must ALWAYS stay in memory.

Contiguous Allocation 
• Main memory must support both OS and user processes 
• Limited resource, must allocate efficiently 
• Contiguous allocation is one early method 
• Main memory usually into two partitions: 
• Resident operating system, usually held in low memory with interrupt vector 
• User processes then held in high memory 
• Each process contained in single contiguous section of memory 

Contiguous Allocation (Cont.) 
• Relocation registers used to protect user processes from each other, and from changing operating-system 
code and data 
• Base register contains value of smallest physical address 
• Limit register contains range of logical addresses — each logical address must be less than the limit register 
• MMLJ maps logical address dynamically 
• Can then allow actions such as kernel code being transient and kernel changing size 

Hardware Support for Relocation and Limit Registers 
limit 
register 
log Cal 
address 
CPU 
trap: addressing error 
relocation 
register 
physical 
address 
memory 

Multiple partition allocation:

Process allocation: when a process is finished or put away, a "hole" is left behind.

Can keep loading processes as long as we have a whole the process can fit into

If we've got a process that's too big for the hole, we might end up with fragmentation.

Process gets split across multiple holes

Multiple-partition allocation 
• Multiple-partition allocation 
• Degree of multiprogramming limited by number of partitions 
• Variable-partition sizes for efficiency (sized to a given process' needs) 
• Hole — block of available memory; holes of various size are scattered throughout memory 
• When a process arrives, it is allocated memory from a hole large enough to accommodate it 
• Process exiting frees its partition, adjacent free partitions combined 
• Operating system maintains information about: 
a) allocated partitions b) free partitions (hole) 
os 
process 5 
process 8 
process 2 
os 
process 5 
process 2 
process 5 
process 9 
process 2 
os 
process 5 
process 9 
process 10 
process 2 

Example: We will apply this for semester tests, exams etc!

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| hole | a | b | c | d | e | f | g |
| size | 13 | 8 | 12 | 30 | 13 | 15 | 17 |

And we have processes of size 8. 10 and 13

First-fit model:

Find the first hole that can fit. Eg 8 would go into a. How much is left? 5!

Then 10 would go into c (2 left) and 13 would go into d, 17 left.

If eg minimum size a process can be is 3 units, then c only has 2 so will never be able to use that hole - "internal fragmentation"

Best-fit model: Find first hole that fits the request the best

So 8 would go into b. First A is our best fit, then b is, then no others are better.

10 would go int c

24 would go into a.

Worst-fit model: allocates largest whole for biggest leftover. Tries to leave the largest holes behind so we can fit more processes

So 8 would go into d, 22 left

10 would go into d again, 12 left over

Then 13 goes into g, 4 left over.

Dynamic Storage-Allocation Problem 
How to satisfy a request of size n from a list of free holes? 
• First-fit: Allocate the first hole that is big enough 
• Best-fit: Allocate the smallest hole that is big enough; must search entire list, unless ordered by size 
• Produces the smallest leftover hole 
• Worst-fit: Allocate the largest hole; must also search entire list 
• Produces the largest leftover hole 
First-fit and best-fit better than worst-fit in terms of speed and storage utilization 

External fragmentation: we have enough space, but it's too spread apart, not continuous

Internal fragmentation:   
both principles still apply to file storage as well

Fragmentation 
• External Fragmentation — total memory space exists to satisfy a request, but it is not contiguous 
• Internal Fragmentation — allocated memory may be slightly larger than requested memory; this size 
difference is memory internal to a partition, but not being used 
• First fit analysis reveals that given N blocks allocated, 0.5 N blocks lost to fragmentation 
• 1/3 may be unusable -> 50-percent rule 

Compaction: Stop all processes and rearrange them.

Only possible if relocation is dynamic and done at execution time, so done while they're sleeping

IO problem addressed by double buffering

Fragmentation (Cont.) 
• Reduce external fragmentation by compaction 
• Shuffle memory contents to place all free memory together in one large block 
• Compaction is possible only if relocation is dynamic, and is done at execution time 
• I/O problem 
• Latch job in memory while it is involved in I/O 
• Do I/O only into OS buffers 
• Now consider that backing store has same fragmentation problems 

Every program has a main function, may have objects, functions etc. Can break up these components and let them live in their own memory locations.  
As a programmer you have to define it that way.

Segmentation 
• Memory-management scheme that supports user view of memory 
• A program is a collection of segments 
• A segment is a logical unit such as: 
main program 
procedure 
function 
method 
object 
local variables, global variables 
common block 
stack 
symbol table 
arrays 

ssajppe —601 
шрјбојсј 
щеш 
atqel 
au!lnojqns 
шејбои е 40 мел s,jesn 

So each of these has different memory requirements and can be mapped differently

Windows uses both paging and segmentation

So in assembly, .data and .code etc are segments. Though because we use .MODEL FLAT we aren't actually using segmenting properly.

Can also have multiple code segments. Just update the register to know what code segment you are referring to.

Logical View of Segmentation 
2 
3 
4 
user space 
4 
2 
3 
physical memory space 

Logical addresses consist of a segment-number and an offset

Segment table maps these 2d physical addresses. Each entry (address) has base + limit (length of segment).

Segmentation Architecture 
• Logical address consists of a two tuple: 
<segment-number, offset>, 
• Segment table — maps two-dimensional physical addresses; each table entry has: 
• base — contains the starting physical address where the segments reside in memory 
• limit — specifies the length of the segment 
• Segment-table base register (STBR) points to the segment table's location in memory 
• Segment-table length register (STLR) indicates number of segments used by a program; 
segment number s is legal if s < STLR 

You can define segments for different purposes, eg code or data.  
And set up different rules, eg data can't execute code, code can't be changed.

Different Read/Write/Execute privledges

Segmentation Architecture (Cont.) 
• Protection 
• With each entry in segment table associate: 
• validation bit = O illegal segment 
• read/write/execute privileges 
• Protection bits associated with segments; code sharing occurs at segment level 
• Since segments vary in length, memory allocation is a dynamic storage-allocation problem 
• A segmentation example is shown in the following diagram 

This type of diagram is important

CPU spits out a tuple: s is segement number, d is offset

S gets the limit and the base address, checks if it's in the allowed range. If it is we can access physical memory

Segmentation Hardware 
CPU 
S 
lint base 
segment 
table 
d 
trap: addræsing error 
physical memry 

Paging 
• Physical address space of a process can be noncontiguous; process is allocated physical memory whenever 
the latter is available 
• Avoids external fragmentation 
• Avoids problem of varying sized memory chunks 
• Divide physical memory into fixed-sized blocks called frames 
• Size is power of 2, between 512 bytes and 16 Mbytes 
• Divide logical memory into blocks of same size called pages 
• Keep track of all free frames 
• To run a program of size N pages, need to find N free frames and load program 
• Set up a page table to translate logical to physical addresses 
• Backing store likewise split into pages 
• Still have Internal fragmentation 

Break up memory address into tuple again, but instead this one has page number + page offset.

Address Translation Scheme 
• Address generated by CPU is divided into: 
• Page number (p) — used as an index into a page table which contains base address of each page in physical memory 
• Page offset (d) — combined with base address to define the physical memory address that is sent to the memory unit 
page number page offset 
p 
• For given logical address space 2m and page size 2n 
d 
n 

We have 16 total bits.

Local address space 65536.

12 offset bits.

6 page number bits.

4096 addresses per page (page size)

64 pages

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 0 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

If we use 1 bit, our memory space can be broken up into 2 pages. 0 or 1

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

Or

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

Think of a long roll of paper, 65536 metres (bits) long. If we tear off 2 (bits) metres, the rest of the memory can be allocated to other stuff

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 0 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

Value of 0

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 1 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

Value of 1

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 0 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

Value of 2

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 1 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

Value of 3

So we can now have 4 pages

If we use 3 bits, we can have 8 pages.

Each page can point to a different place in memory.

Number of bits in the address tells us how many pages we can have.

So p takes us to a page table which lets us do a lookup. We then get back a "frame number" which we prepend to the offset address.

Paging Hardware 
foooo 0000 
CPU 
address 
physical 
address 
physical 
page table 

So an actual exercise:

Given:

Convert the following 8bit logical adderss into an 8 bit physical address

|  |  |  |
| --- | --- | --- |
| index | Page frame | valid |
| 3 | 00 | 1 |
| 2 | 11 | 1 |
| 1 | 01 | 1 |
| 0 | 10 | 1 |

((So we're using 2 bits because 4 different answers))

Local address: 129

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Our calculations:

Converted local address to binary:

1000001

He got 10 by looking at the 2 most significant bits. (2 bits because page table has 2^2 aka 4 answers, so 2 bits)

So he got page number of 2 (10 in binary)

There's an offset of 000001 (what's left after the significant bits) (not used in calculations)

Then got the frame number 11 by looking at the table at index 2

So then converted to binary physical by replacing those significant bits with what we got from the table

1100 0001

And converted back to decimal 193

another example:

Logical address; 64:

So binary logical is 100000

So the offset is 00000

The page number is 01 (because look at binary logical as 8 bits, add a 0 in front) so look at index 1 on the table, get 01

The physical address is then 01000000 (happens to be the same in this case)

And then convert to decimal again

Another example:

Logical address was 1:

Binary 00000001

Page number is 00, the table tells us to return 01

So then the physical address is 01000001

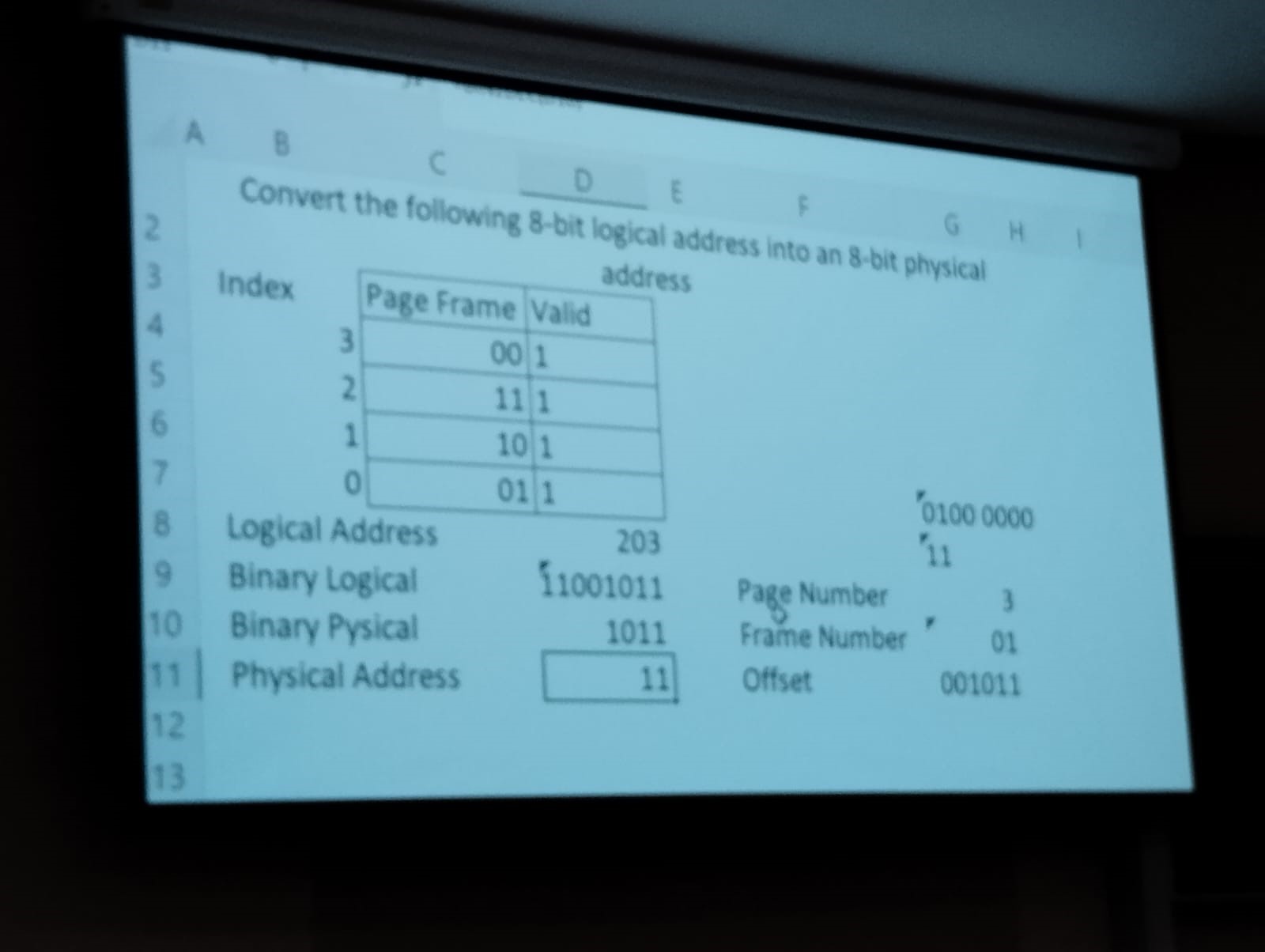
Decimal-ified is 65

Another example: logical address 203

Binary it's: 203 - 128 = 75 - 64 = 11 -8 = 3 -2 = 1

So 203 = 2^0 + 2^1 + 2^3 + 2^6 + 2^7 = 1100 1011

So 11



|  |  |
| --- | --- |
| (Each can hold 100 pages/addresses) | Frame |
| 8 |  |
| 7 | 6 |
| 6 | 7 |
| 5 | 1 |
| 4 | 2 |
| 3 | 0 |
| 2 | 3 |
| 1 | 5 |
| 0 | 4 |

|  |  |  |  |
| --- | --- | --- | --- |
| PageNumber (start here) | Index (significant digit) | Frame | Offset (at frame 4, 25th page) (He did this as a physical pages in lesson) |
| 025 | 0 | 4 | 25 |
| 114 | 1 | 5 | 14 |
| 221 | 2 | 5 | 14 |
| 420 | 4 | 2 | 20 |

If we rarely read pages from eg Frame 4, we don’t have to keep it in the table.

**Implementation of Page Table**

Each process has its own page table.

Page table is stored in main memory.

Pages fit in memory, then update page table.

It is not very efficient, as 2 lookups are needed.

TO get around this, there is a **translation lookup buffer(TLB).  
-TLB:** A buffer just for the page lookups. It caches the frame numbers/address. (as each page(?) is 4mb, which is 4 xxx amount of potential addresses)   
-TLB miss: When frame not in TLB, add it to TLB. (Kinda like the bonus array in our prac 6 recursion)

**Memory protection**

To stop a lookup from looking at memory out of memory bounds (marks frame as **invalid**).

**Shared Pages**

Shared Code: One copy of a read only (**reentrant)** code is shared among processes (eg text editors).

Each process keeps a separate data section. 1 copy of code (advantage).

**Structure of Page tables**

Another **disadvantage of page tables** is that they can grow quite big and take up a lot of memory.

You can solve this problem with 3 things: (These 3 **are not important – wont be tested on at all)**

**1.Hierarchical Paging:**

You have 2 levels of tables.

You have an index which points to a secondary table, and at that table it points to the page you want.

This decreases size, as secondary tables (inactive ones) can be in storage instead of Ram.

**2.Hashed page tables**

The logical address (page frame -p-) is hashed, and hopefully is unique enough to not have too many collisions.

This doesn’t decrease size, but increases speed (if low collisions).

**3.Inverted Page Tables**

Draws a map of the physical memory, as it is always the same size. So you have one page table that describes a eg 1GB of RAM.

This is slow. To speed this up, combine this with hash