



COMP 273

Virtual Memory

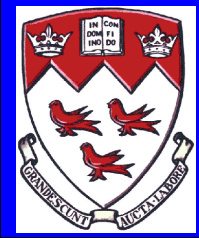
Part 1

Prof. Joseph Vybihal



Announcements

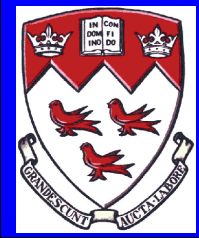
- Course evaluation
- Today: Last class
- Exam
 - Monday, Dec 8, 2PM
 - Old exam posted
 - Tutorials by TA s – check myCourses





Part 1

Virtual Memory Basics





Standard MIPS Memory

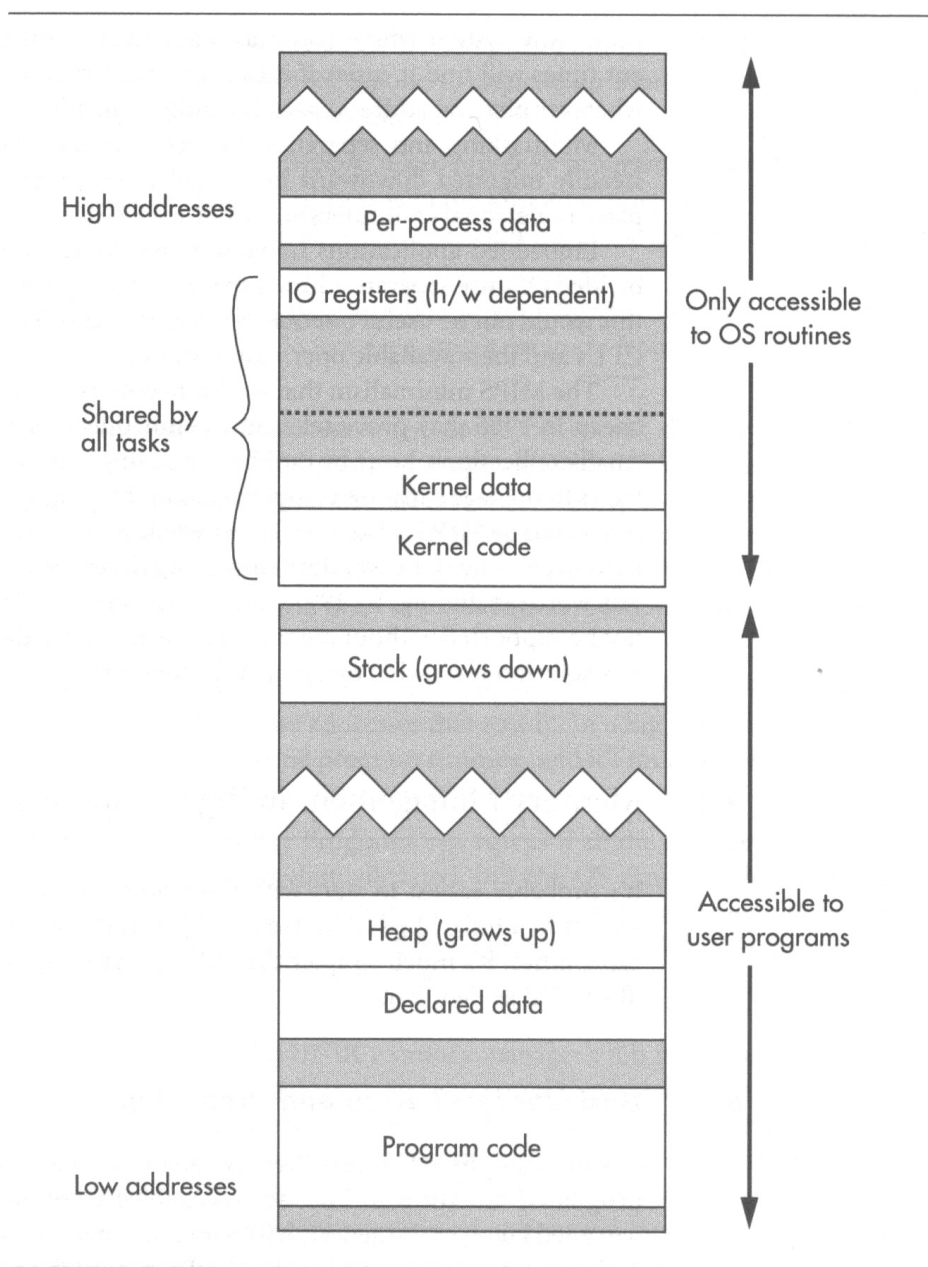


FIGURE 6.1 Memory map for a protected process



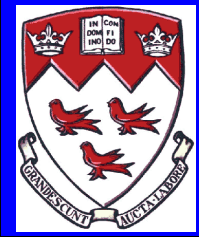
Motivation for VM

- A simulator for memory giving your computer the impression that it has more RAM.
- Removes the burden from a programmer in managing limited RAM.
- VM helps to allow multiprocessing by simulating more space in RAM



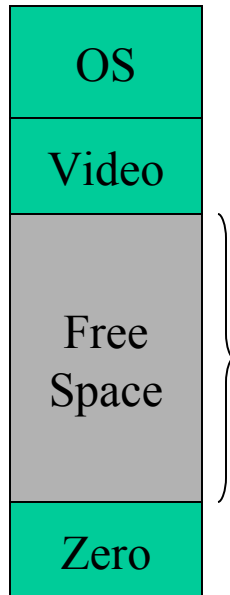
Memory Management Types

- None
- None with cheating by programmer
 - Terminate Stay Resident (TSR)
- Compiler Managed
 - Overlaying
- OS Managed
 - Page swapping & Virtual Memory



Classical Software Development:

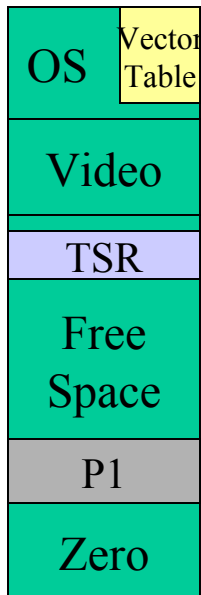
- Program compiled with no special features
- Linker adds a loader to the executable
- Loader inserts code into free space at a given start address
- OS notified of its existence
- Program executes to completion then terminates
- OS notified of termination and removed from RAM





Management By: Programmer (cheating)

Memory Type : TSR



Software Development:

- Compiled with OS notification turned off
- Linker adds a loader to the executable
- Loader inserts code into free space at a given start address
- OS NOT notified of its existence
- Program executes
 - Modifies the OS interrupt vector (point to itself)
 - Then it terminates
- OS NOT notified of termination
 - Program NOT removed from RAM
- A subsequent program loaded into RAM can use TSR
 - Uses Interrupt Vector to switch to TSR and back

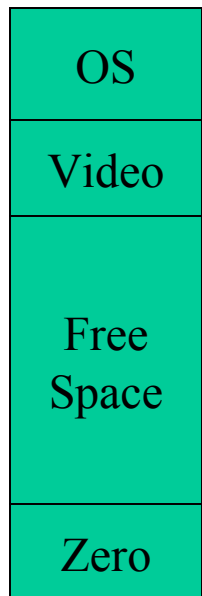
Terminate and Stay Resident





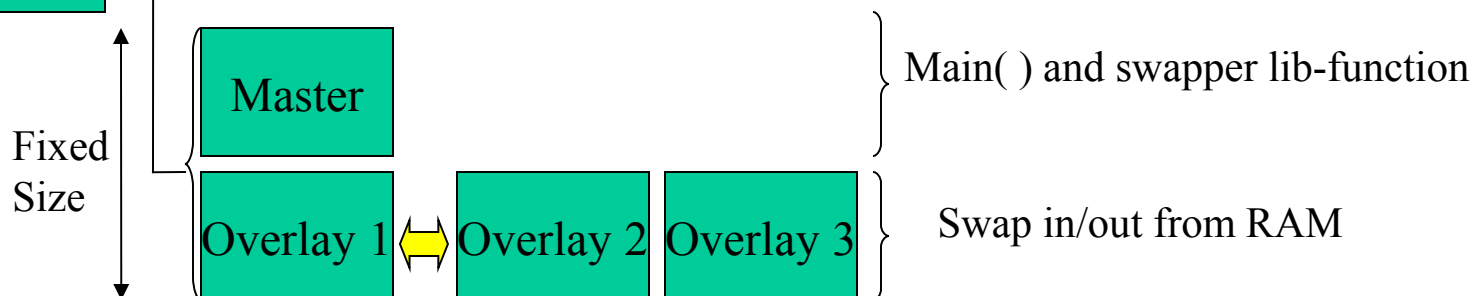
Management By: Compiler

Memory Type : Overlay

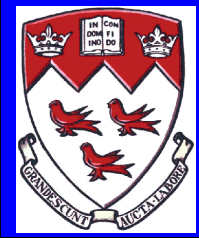


Software Development:

- Compiled with overlay mode ON
- Program compiled into fixed sized OVERLAYS
- Each overlay can be loaded and run independently in RAM
- OS notified of complete program's existence
- Program executes to completion then terminates
 - Master and slave frames
- OS notified of termination and removed from RAM



Permits very large programs to run in smaller RAM

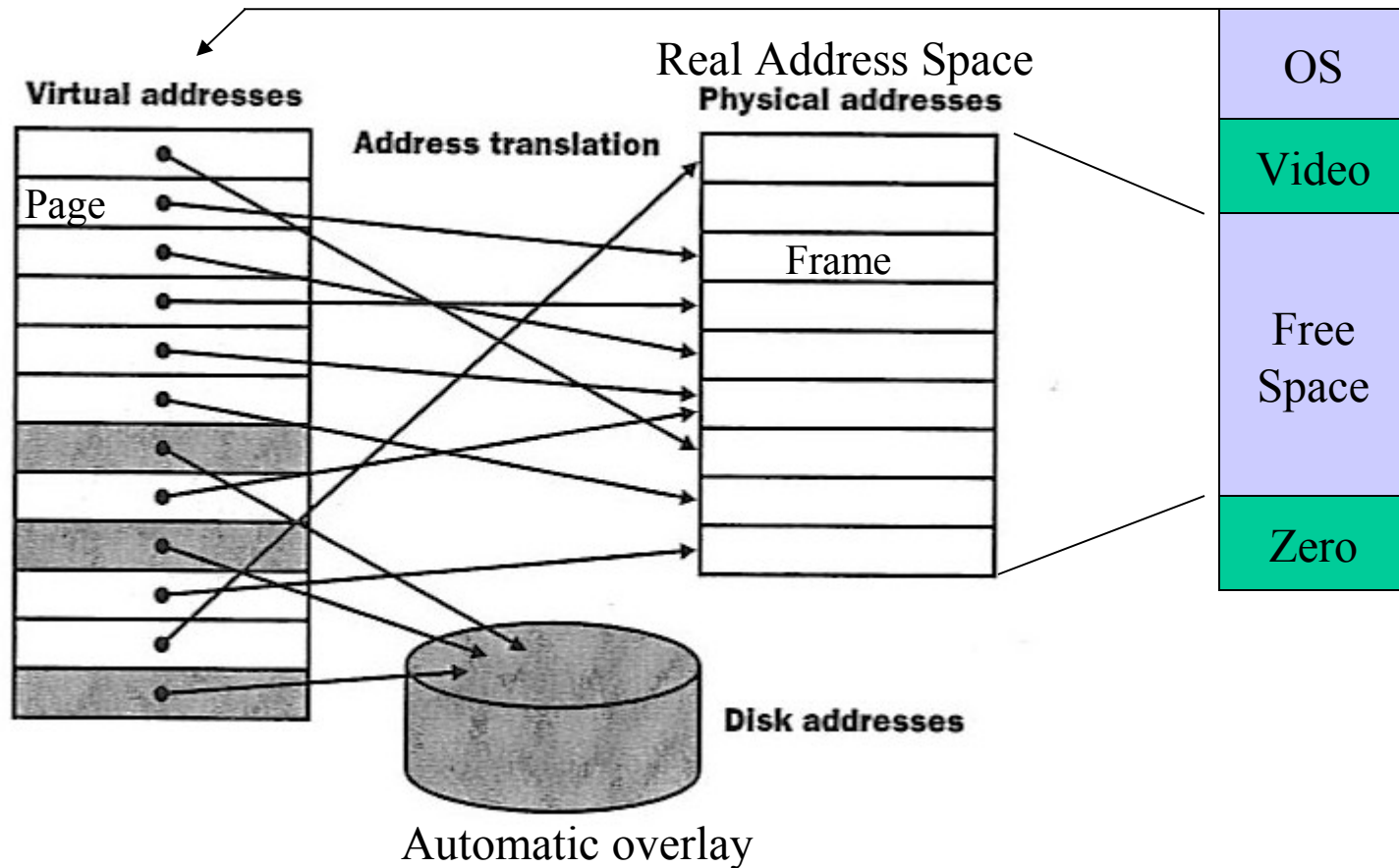




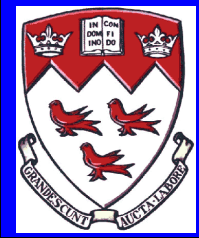
Management By: OS

Memory Type : Virtual Memory

Virtual Address Space



Permits very large programs to run in smaller RAM





Virtual Memory Method

1. Launch a new program

1. Convert it into a process
2. Convert the code into pages
3. All pages “virtually” loaded into VM
4. A subset of pages actually loaded into RAM (Frames)
5. Memory map between VM and RAM

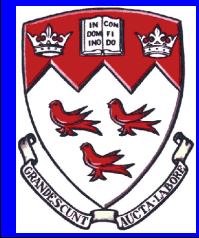
2. Execute program

1. Instructions execute until end of page
2. Search for next page in RAM
3. If not found → “Page Fault” → do overlay
4. Continue executing program from new loaded page



VM Properties

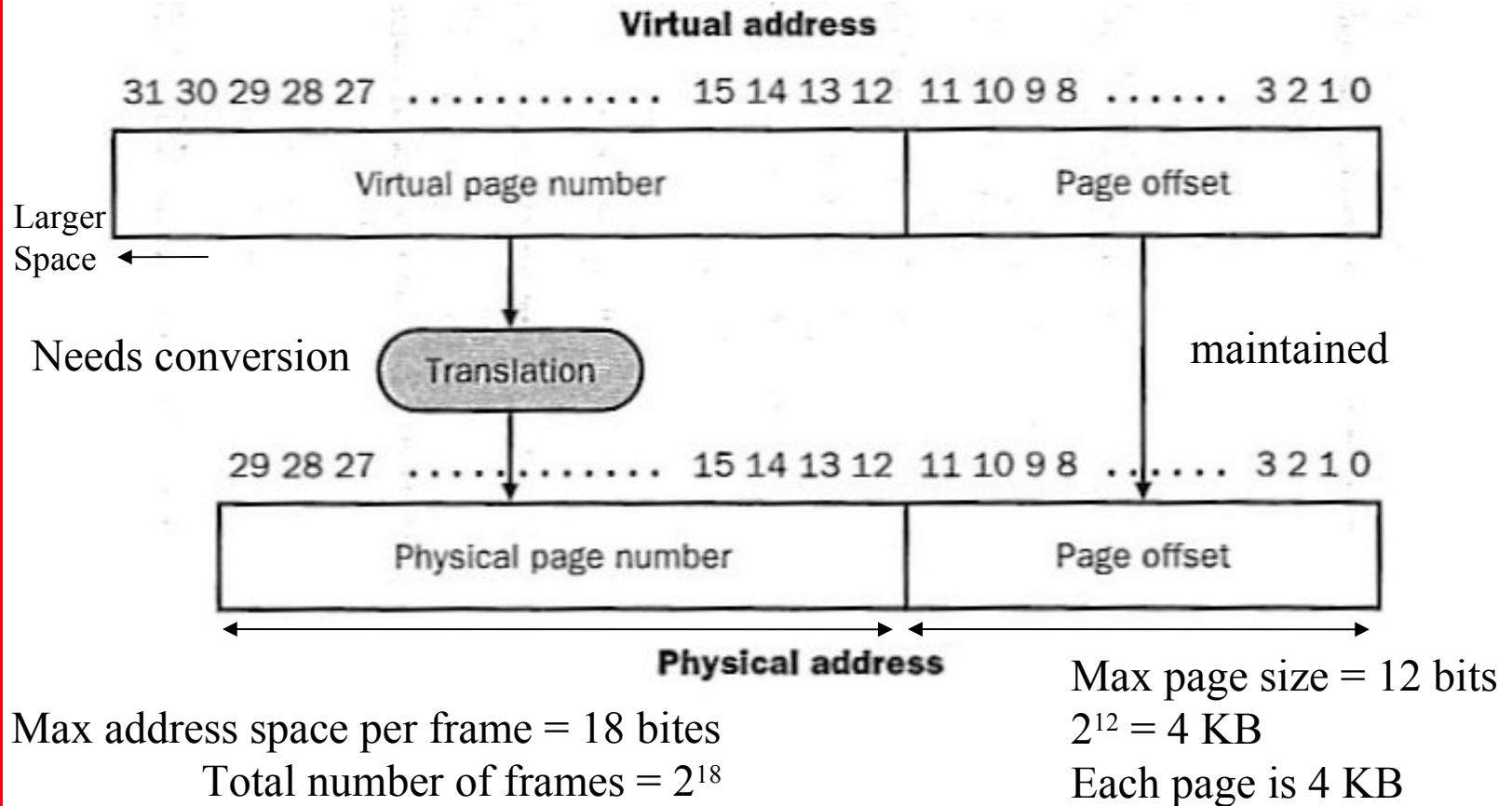
- Programmer addresses code based on VM address
- OS, therefore, constructed to manages code from VM space (from that point of view)
- BUT, code must actually execute on real hardware = RAM and CPU
- THEREFORE, need to convert all addresses to real RAM values
- This must be a fast process
- This must take into account the page / frame duality of this technique



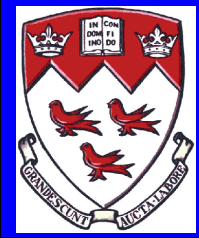
Part 2

Virtual Memory Specifics

Mapping from VM to RAM



Therefore: RAM = $2^{18} * 2^{12} = 1 \text{ GB}$ (frames)
VM = $2^{20} * 2^{12} = 4 \text{ GB}$ (pages)





Remember

Storage	Technology	Speed	Cost
CPU Registers	Flip-flops	1 – 5 ns	\$250 - \$300
Cache	SRAM	5 – 25 ns	\$100 - \$250
RAM	DRAM	60 – 120 ns	\$5 - \$10
Disk	Magnetic charge – mechanical	10 Million ns – 20 million ns	\$0.1 - \$0.2

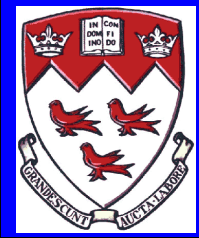


~ per 1 Meg



Huge Cost of Page Faults

- Page faults imply load overlay from disk!
- How big should a page be?
 - Amortization of disk access time
 - Common page sizes: 4, 16, 32, 64 KB
- OS driven, therefore algorithmic selection:
 - Page loading order
 - Disk drive considerations





Page Loading Order

- On demand
 - When we need a page get the overlay from disk
 - What if all the frames are filled? Which one do we overlay? (*the victim*)
- Is there a best overlay selection procedure?
 - Least Recently Used frame
 - First Come First Serve frame replacement
 - Replace all frames of another process
 - Randomly select a frame and overlay



Disk Drive Considerations

- Dirty Pages
 - A page in RAM whose data has changed has been selected to be overlaid
 - Need to write that page back to disk (**write-back**)
 - Read in new page becomes also write out old page
- Byte or Block Disk Access
 - Which is faster?
 - Merging block with buffer & frame improves speed
 - Seek time for next byte is skipped
 - For block we increment pointer after first seek
 - Using DMA also good
 - Using Interrupts also good



Two Types of VM

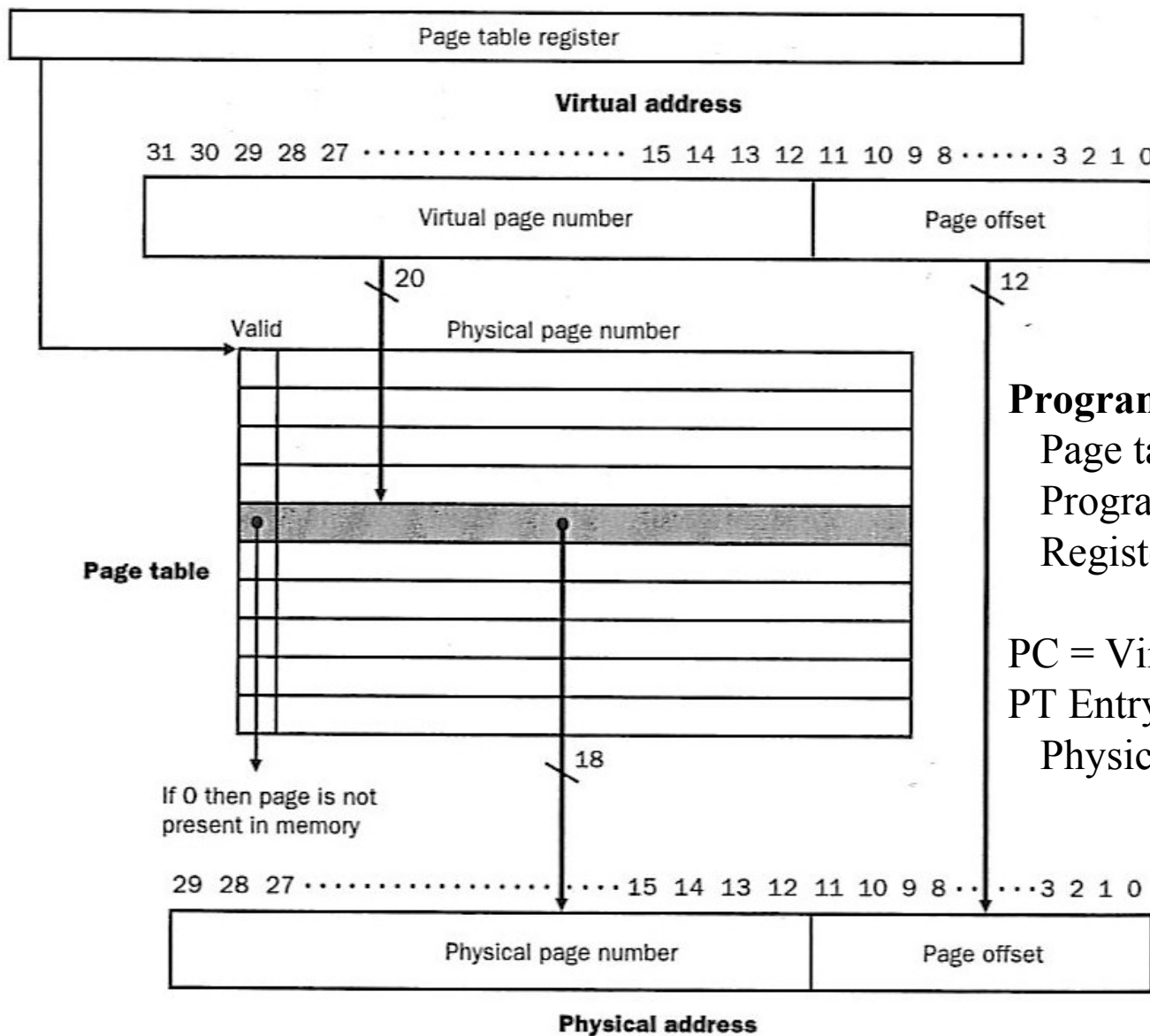
- Paging
 - Fixed size overlays
 - Overlay matches frame size
 - Addressing:

Page #	Offset
--------	--------
- Segmentation
 - Variable sized overlays
 - Overlays are multiples of frame size
 - This is true for simplicity
 - Can also be implemented with true variableness
 - Addressing:

Segment #	Page #	Offset
-----------	--------	--------



Paging Hardware

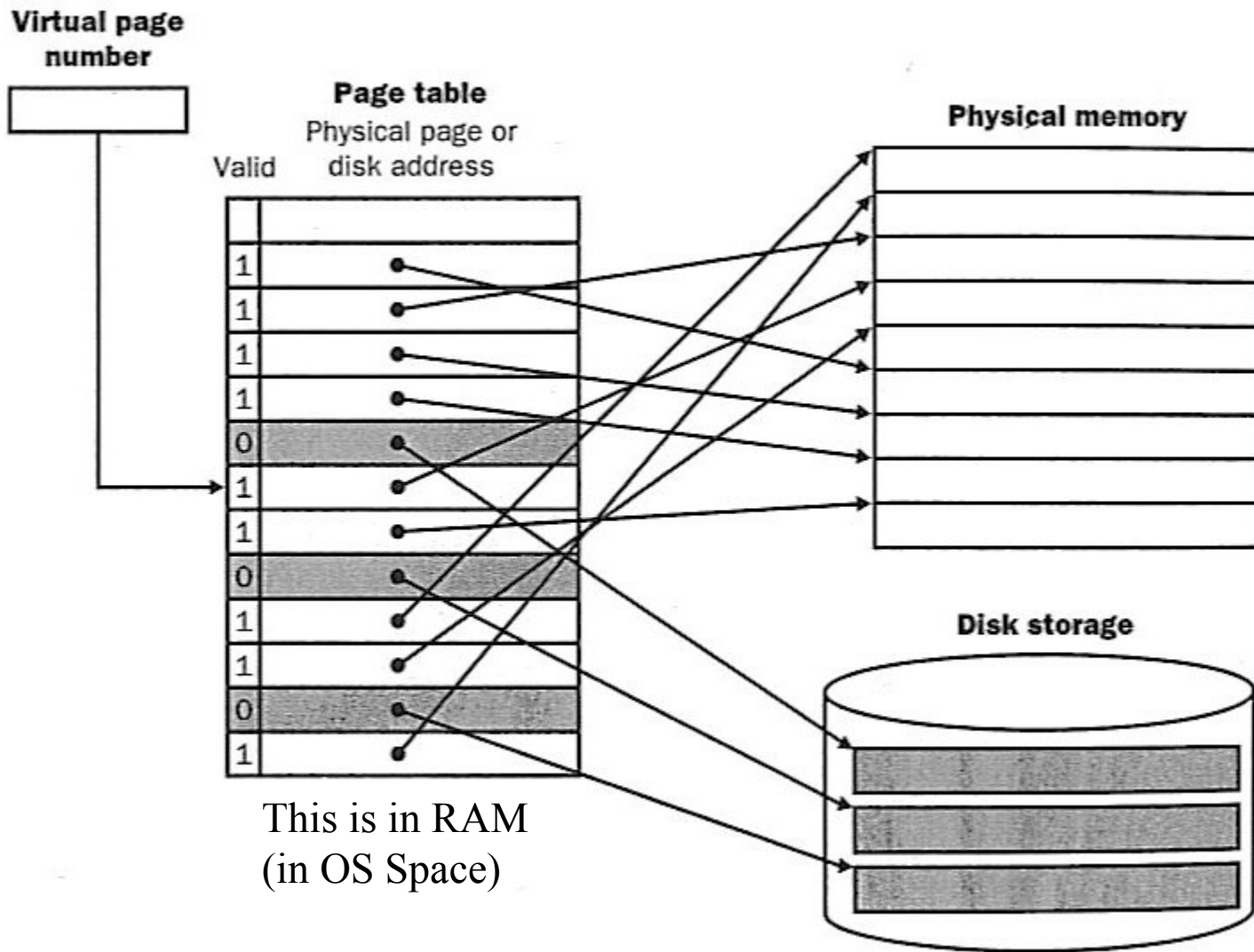


Program State =
Page table entry +
Program counter +
Registers

PC = Virtual Address
PT Entry =
Physical page number

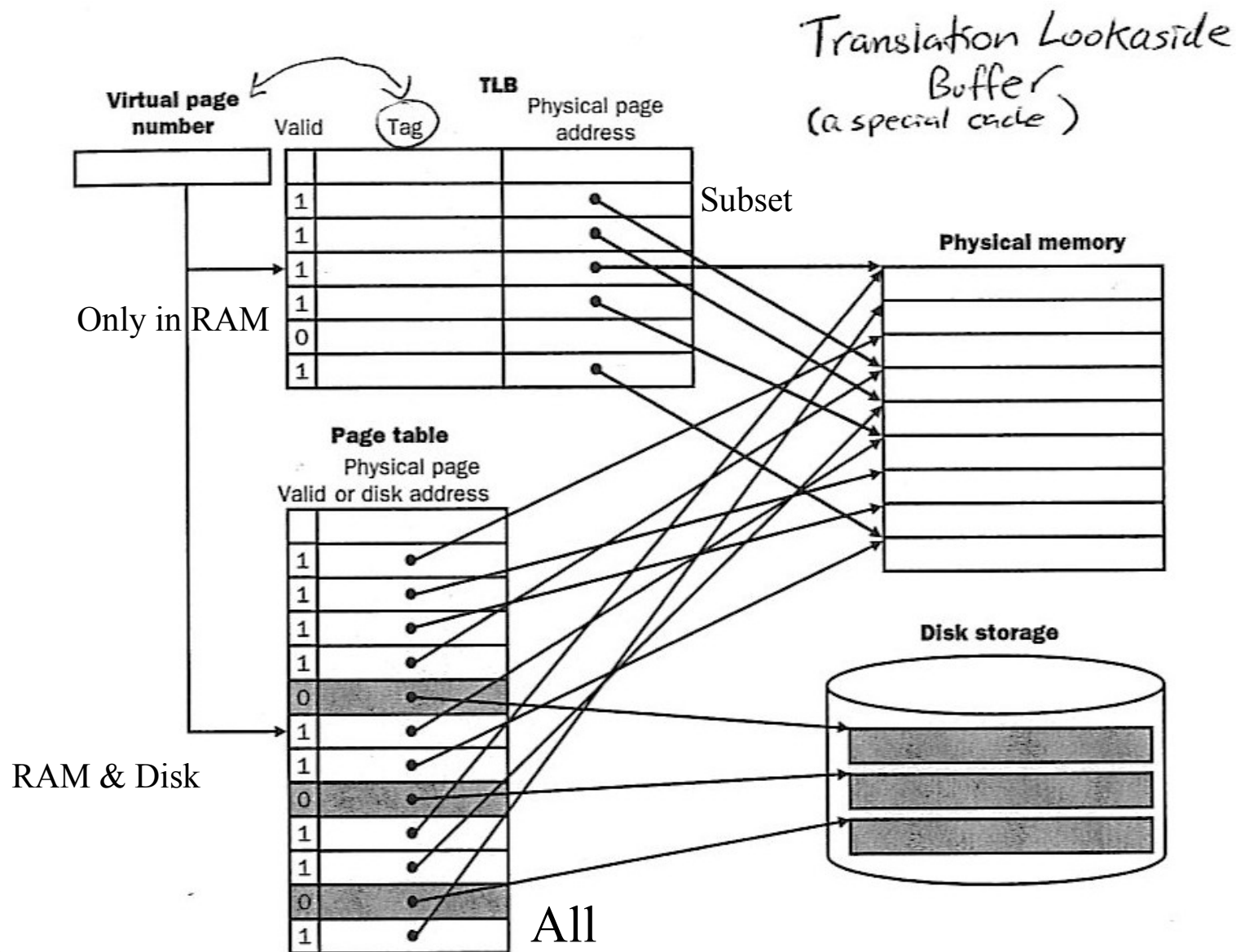


The Page Table



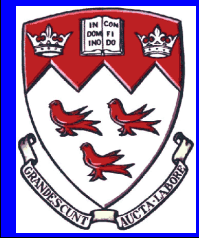
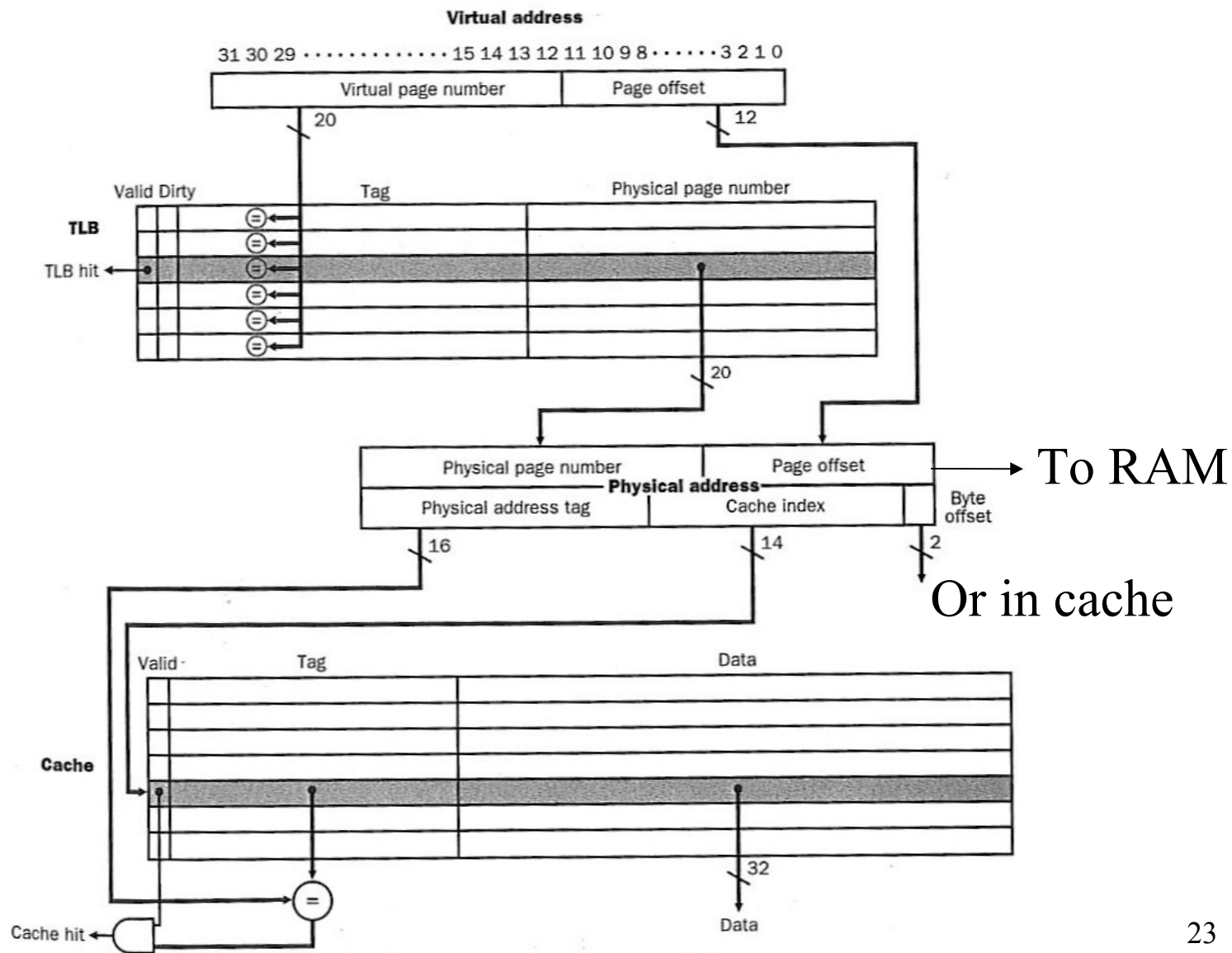


Faster Address Translation



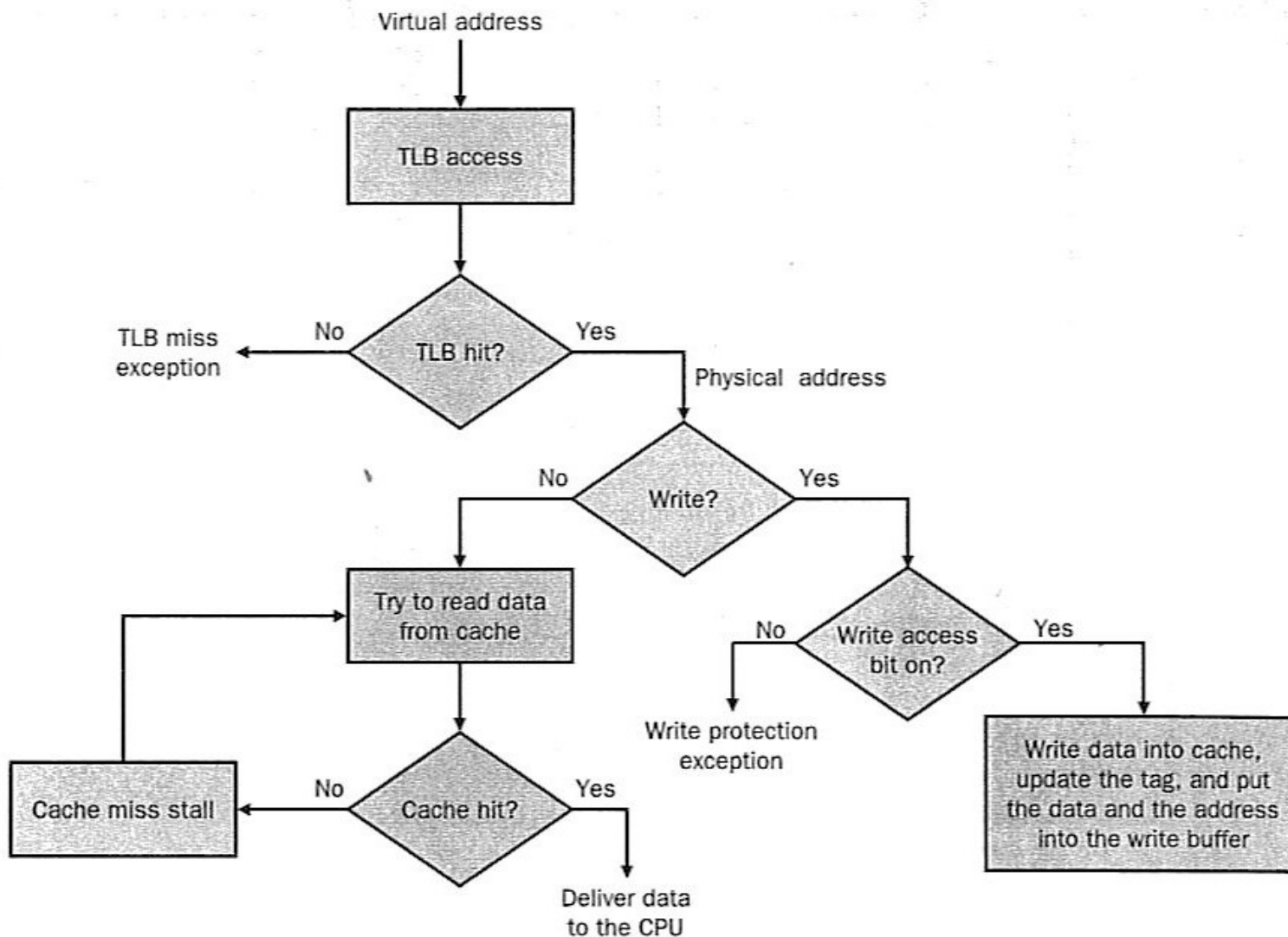


Implementation



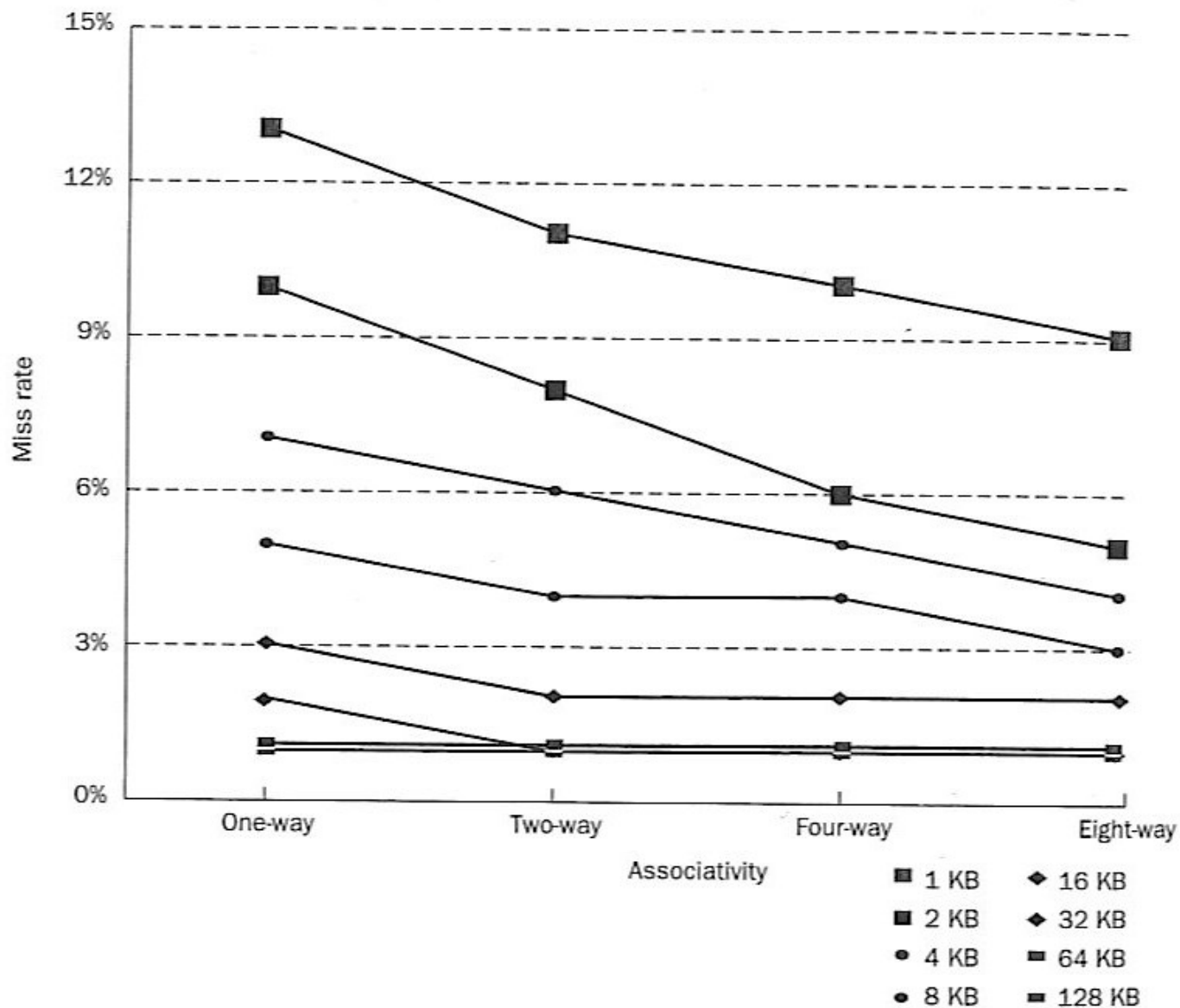


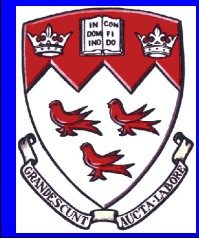
Flowchart





Statistics





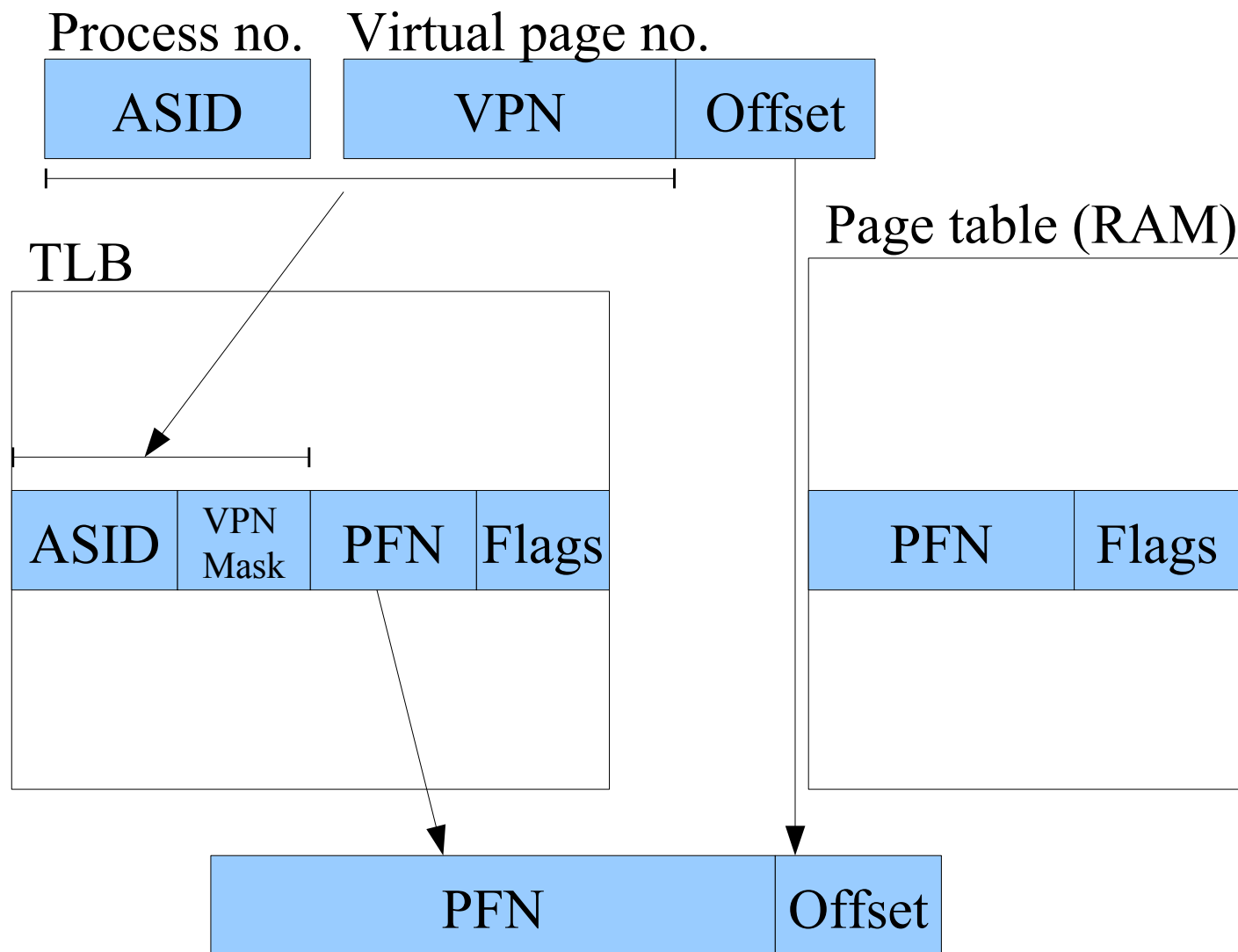
Part 3

The SPIM MMU (Memory Management Unit)

(Optional Material for student to read – not covered in class)

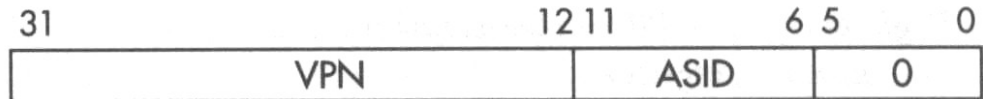


Memory Translation System

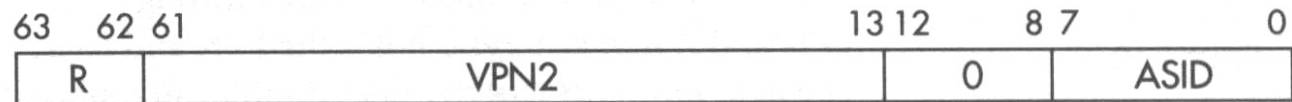


Co-Processor 0 MMU Registers

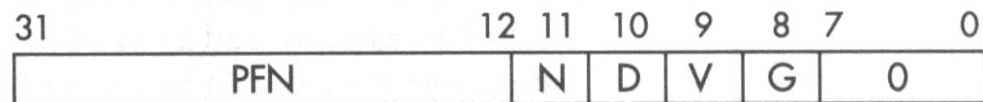
EntryHi register (TLB key fields) R3000-style CPUs



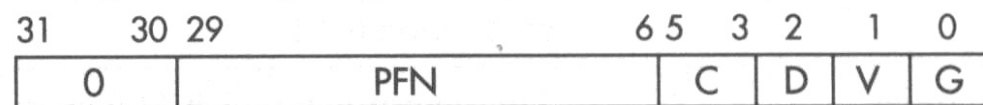
EntryHi register (TLB key fields) R4000-style CPUs



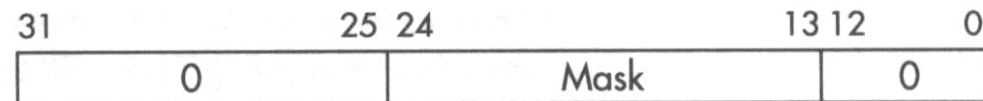
EntryLo register (TLB data fields) R3000-style CPUs



EntryLo0,1 register (TLB data fields) R4000-style CPUs



PageMask register 64-bit CPUs only



Notes:

VPN, virtual page number

ASID, address space id

R, address region

PFN, VPN high order bits

N, non-cacheable

C, cache algorithm

D, dirty bit write enable

V, valid boolean

G, global address shared

0, zeros



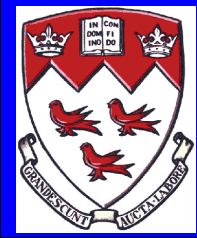
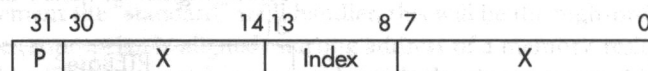


TABLE 6.1 CPU control registers for memory management

Register mnemonic	CP0 register number	Description
EntryHi	10	<p>Together these registers hold everything needed for a TLB entry. All reads and writes to the TLB must be staged through them. EntryHi holds the VPN and ASID; EntryLo holds the PFN and flags.</p> <p>The field EntryHi (ASID) does double duty, since it remembers the currently active ASID.</p> <p>In some CPUs (all 64-bit CPUs to date) each entry maps two consecutive VPNs to different physical pages, specified independently by two registers called EntryLo0 and EntryLo1.</p> <p>EntryHi grows to 64 bits in 64-bit CPUs but in such a way as to preserve the illusion of a 32-bit layout for software that doesn't need long addresses.</p> <p>PageMask can be used to create entries that map pages bigger than 4KB; see Section 6.3.1.</p>
EntryLo/EntryLo0	2	
EntryLo1	3	
PageMask	5	
Index	0	This determines which TLB entry will be read/written by appropriate instructions.
Random	1	This pseudo-random value (actually a free-running counter) is used by a tlbwr to write a new TLB entry into a randomly selected location. Saves time when processing TLB refill traps, for software that likes the idea of random replacement (there is probably no viable alternative).
Context	4	<p>These are convenience registers, provided to speed up the processing of TLB refill traps. The high-order bits are read/write; the low-order bits are taken from the VPN of the address that couldn't be translated.</p> <p>The register fields are laid out so that, if you use the favored arrangement of memory-held copies of memory translation records, then following a TLB refill trap Context will contain a pointer to the page table record needed to map the offending address. See Section 6.3.5.</p> <p>Xcontext does the same job for traps from processes using more than 32-bits of effective address space; a straightforward extension of the Context layout to larger spaces would be unworkable because of the size of the resulting data structures. Some 64-bit CPU software is happy with 32-bit virtual address spaces, but for when that's not enough 64-bit CPUs are equipped with "mode bits" SR(UX), SR(KX) which can be set to cause an alternative TLB refill handler to be invoked; in turn that handler can use Xcontext to support a huge but manageable page table format.</p>
Xcontext	20	



MIPS I CPUs



All MIPS III and higher CPUs to date

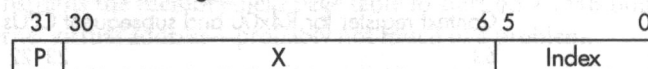
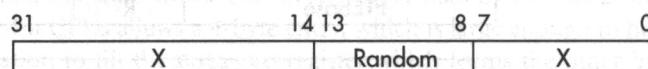


FIGURE 6.5 Fields in the **Index** register

32-bit CPUs to date



64-bit CPUs to date

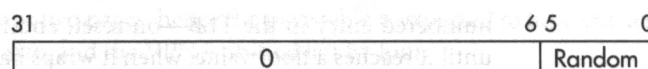


FIGURE 6.6 Fields in the **Random** register

Notes:

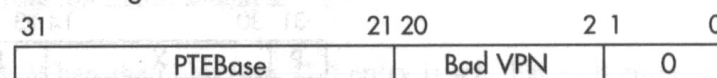
- P, valid index found bool
- X, address value
- Index, TLB position
- Random, random index (auto)
- PTEBase, start page table ptr
- Bad VPN, address exception ptr
- 0, zeros

PageMask bits

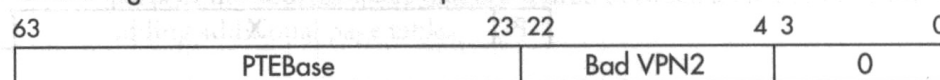
Page size

24-21	20-17	16-13	
0000	0000	0000	4KB
0000	0000	0011	16KB
0000	0000	1111	64KB
0000	0011	1111	256KB
0000	1111	1111	1MB
0011	1111	1111	4MB
1111	1111	1111	16MB

Context register for R3x00 CPUs



Context register for R4x00 and subsequent CPUs



XContext register for R4x00 and subsequent CPUs only

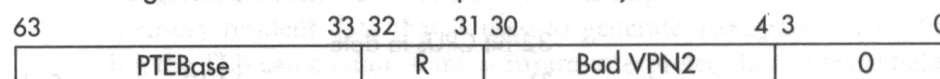


FIGURE 6.7 Fields in the **Context/XContext** registers



MMU Control Instructions

- `tlbr` # read TLB entry into index
- `tlbwi` # write TLB entry from index
- `tlbwr` # write TLB entry selected randomly
- `tlbp` # TLB lookup (uses VPN & ASID)



Code Example

```
#include <mips/r3kc0.h>

LEAF(mips_init_tlb)
    mfc0      t0,C0_ENTRYHI          # save ASID
    mtc0      zero,C0_ENTRYLO        # tlblo = !valid
    li        a1,NTLBID<<TLBIDX_SHIFT # index
    li        a0,KSEG1_BASE          # tlbhi = impossible VPN

    .set noreorder
1:  subu      a1,1<<TLBIDX_SHIFT
    mtc0      a0,C0_ENTRYHI
    mtc0      a1,C0_INDEX
    addu      a0,0x1000              # increment VPN, so all entries differ
    bnez      a1,1b
    tlbwi                      # in branch delay slot
    .set      reorder

    mtc0      t0,C0_ENTRYHI          # restore ASID
    j         ra
END(mips_init_tlb)
```

TLB Initialization



TLB Exception Code

```
.set      noreorder
.set      noat

TLBmissR3K:
    mfc0   k1,C0_CONTEXT      # (1) Get address of page table
    mfc0   k0,C0_EPC          # (2) Get return address
    lw     k1,0(k1)           # (3) Get contents pointed to
    nop                                # (4) Wait, load takes 2 clock ticks
    mtc0   k1,C0_ENTRYLO      # (5) LO = k1, Hi auto loaded
    nop                                # (6) Wait again...
    tlbwr                                # (7) Write randomly to TLB
    jr     k0                  # (8) Return to user program
    rfe                                # (9) Delay slot execution...
                                restore CPU state in SR
.set      at
.set      reorder
```

