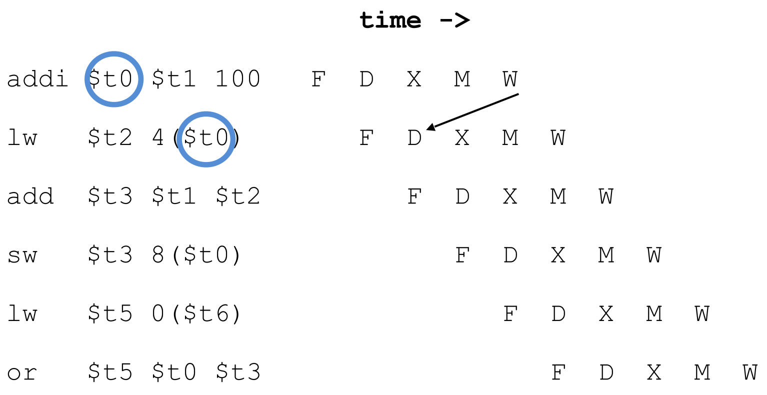
Pipelining Exercises, Continued

1. Spot all data dependencies (including ones that do not lead to stalls). Draw arrows from the stages where data is made available, directed to where it is needed. Circle the involved registers in the instructions. **Assume no forwarding.** One dependency has been drawn for you.



```
                                  time ->

addi $t0 $t1 100    F  D  X  M  W
lw   $t2 4($t0)        F  D  X  M  W
add  $t3 $t1 $t2          F  D  X  M  W
sw   $t3 8($t0)              F  D  X  M  W
lw   $t5 0($t6)                 F  D  X  M  W
or   $t5 $t0 $t3                   F  D  X  M  W
```

2. Redraw the arrows for the above question assuming that **our hardware provides forwarding**.

```
                                  time ->

addi $t0 $t1 100    F  D  X  M  W
lw   $t2 4($t0)        F  D  X  M  W
add  $t3 $t1 $t2          F  D  X  M  W
sw   $t3 8($t0)              F  D  X  M  W
lw   $t5 0($t6)                 F  D  X  M  W
or   $t5 $t0 $t3                   F  D  X  M  W
```

3. How many stalls will we have to add to the pipeline to resolve the hazards in Exercise 4? How many stalls to resolve the hazards in Exercise 5?

Virtual Memory Overview

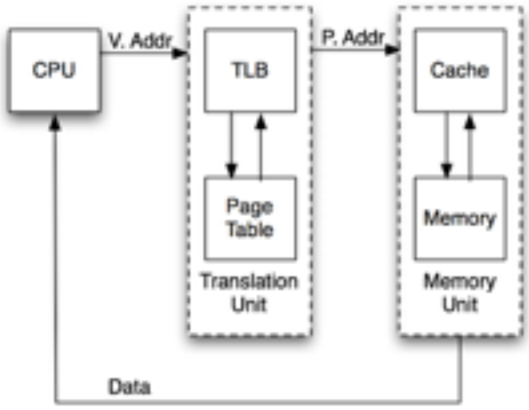
Virtual address (VA): What your program uses

Virtual Page Number	Page Offset

Physical address (PA): What actually determines where in memory to go

Physical Page Number	Page Offset

With 4 KiB pages and byte addresses, 2^(page offset bits) = 4096, so page offset bits = 12.



The Big Picture: Logical Flow
Translate VA to PA using the TLB and Page Table. Then use PA to access memory as the program intended.

Pages
A chunk of memory or disk with a set size. Addresses in the same virtual page get mapped to addresses in the same physical page. The page table determines the mapping.

The Page Table

Index = Virtual Page Number (not stored)	**Page Valid**	**Page Dirty**	**Permission Bits (read, write, ...)**	**Physical Page Number**
0				
1				
2				
…				
(Max virtual page number)				

Each stored row of the page table is called a **page table entry** (the grayed section is the first page table entry). The page table is stored *in memory*; the OS sets a register telling the hardware the address of the first entry of the page table. The processor updates the "page dirty" in the page table: "page dirty" bits are used by the OS to know whether updating a page on disk is necessary. Each process gets its own page table.

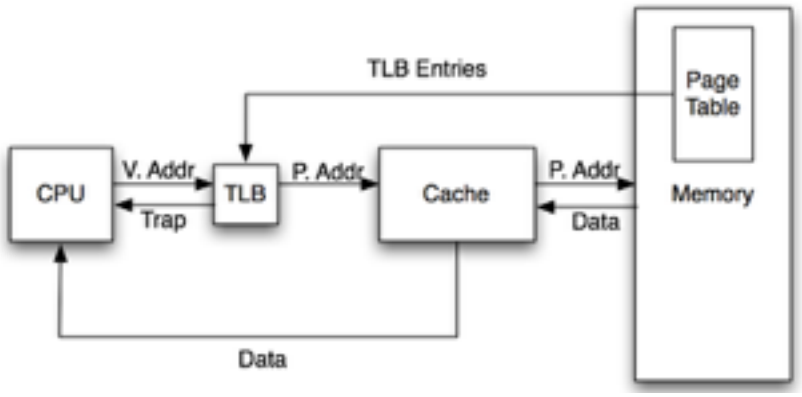
- **Protection Fault**--The page table entry for a virtual page has permission bits that prohibit the requested operation
- **Page Fault**--The page table entry for a virtual page has its valid bit set to false. The entry is not in memory.

The Translation Lookaside Buffer (TLB)

A cache for the page table. Each block is a single page table entry. If an entry is not in the TLB, it's a TLB miss. Assuming *fully associative*:

TLB Entry Valid	Tag = Virtual Page Number	Page Table Entry		
		Page Dirty	Permission Bits	Physical Page Number
…	…	…	…	…

The Big Picture Revisited



Exercises

What are three specific benefits of using virtual memory? [there are *many*]

What should happen to the TLB when a new value is loaded into the page table address register?

x86 has an "accessed" bit in each page table entry, which is like the dirty bit but set whenever a page is used (load *or* store). Why is this helpful when using memory as a cache for disk?

Fill this table out!

Virtual Address Bits	Physical Address Bits	Page Size	VPN Bits	PPN Bits	Bits per row of PT (4 extra bits)
32	32	16KB			
32	26			13	
	32		21		21
		32KB	25		25
64			48		28