

1 Quick Reference

- **Virtual memory:** lets multiple processes run location-independently and protected from each other, and uses main memory as a cache for disk.
- **Segments (base + bound):** addresses are relative to a base register and checked against a bound register; whole segments are evicted to disk.
- **External fragmentation:** free space is not contiguous (holes between segments), so a job may not fit even when enough total memory is free.
- **Page table:** maps each fixed-size virtual page (VA = page number | offset) to a physical page, so contiguous virtual memory can use non-contiguous physical pages.
- **Internal fragmentation:** a process's size is not a multiple of the page size, so part of its last page is unused.
- **Page fault:** the page table walk finds no valid PTE (page swapped out, not yet committed, copy-on-write, etc.). Usually not an error; must be a restartable exception.
- **Hierarchical page table:** the VPN is split into per-level indices (Sv32: 10 | 10 | 12-bit offset); the root is in satp. A **page table walk** is one sequential memory access per level.
- **TLB:** small, typically fully associative cache of PTEs (V/R/W/D bits, VPN tag, PPN). A TLB miss triggers a page table walk; it is not the same as a page fault.

2 CPI Equation Correction

The CPI equation given in Discussion 2, (total cycles after first commit) / (# committed instructions — 1), is **not** the definition used in this class. Per Lecture 3 (Pipelining), CPI includes the first instruction's commit cycle:

$$\text{CPI} = (\text{last commit cycle} - \text{first commit cycle} + 1) / (\# \text{ committed instructions})$$

The Discussion 2 version starts counting the cycle *after* the first commit and therefore also drops the first instruction from the count; the class definition counts from the first commit cycle inclusive and divides by all N committed instructions. Both give 1 for a no-stall sequence, but they differ on when to start counting the CPI. (See Ed #149.)

Below is the completed interlocked pipeline diagram from Discussion 2 for the linked-list traversal program (write-through register file, branches always predicted taken and squashed if wrong, single-cycle memory, 2 loop iterations).

```
      ADDI  x2, x0, 0x700
LOOP: ADD   x1, x2, x0
      LW    x2, 4(x2)
      BNE   x2, x0, LOOP
      SW    x3, 4(x1)
```

Corrected CPI walkthrough: interlock pipeline (given)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
addi	F	D	X	M	W																									
(1) add		F	D	D	D	X	M	W																						
lw			F	F	F	D	X	M	W																					
bne						F	D	D	D	X	M	W																		
(2) add							F	F	F	D	X	M	W																	
lw										F	D	X	M	W																
bne											F	D	D	D	X	M	W													
(3) add												F	F	F	D	-														
lw															F	-														
sw																F	D	X	M	W										

- How many bubbles were needed?

8; 6 from hazards (2 each for the first add, the first bne, and the second bne) and 2 from the branch misprediction.

- What is the CPI for this program with interlocking? (CPI is measured from when the first instruction commits to when the last instruction in the sequence commits.)

$$(\text{Last commit} - \text{first commit} + 1) / (\# \text{ committed instructions}) = \frac{20-5+1}{8} = \frac{16}{8} = 2$$

Ideally the 8 committed instructions would commit in 8 consecutive cycles ($12 - 5 + 1$). With 8 bubbles they take 16 cycles, so 2 cycles per instruction.

Sanity check: ideal + bubbles = $8 + 8 = 16 = \text{total cycles}$.

- What is the CPI assuming branches are always predicted correctly?

$$2 \text{ bubbles come from the branch misprediction, so } \frac{20-5+1-2}{8} = \frac{14}{8} = 1.75$$

- Follow-up: why is the lw held in the fetch stage and not the decode stage, as prescribed by our definition of interlocking?

Only one instruction can occupy a stage at a time. The add is stalled in D, so the lw behind it cannot advance out of F and is held there (the PC is not incremented while the pipeline is stalled). The interlock decision is still made in D; the lw is just backed up behind the instruction that is stalling.

3 Page Size Trade-Offs

- 3.1 What are the benefits of a larger page size? What are the benefits of a smaller page size?

Larger: Smaller page tables; fewer page table walks required.

Smaller: Less internal fragmentation; lower page fault penalty (less traffic to load a new page on a fault).

4 Linear vs Hierarchical Page Tables

- 4.1 Consider 4 GiB (32-bit) of addressable virtual memory, 4 KiB pages, and 4-byte PTEs.

Bits in the page offset: $\log_2(\text{page size in bytes}) = \log_2(4096) = 12$

Bits in the virtual page number: $\text{size of memory address} - \text{page offset bits} = 32 - 12 = 20$

Number of pages: 2^{20} pages

- 4.2 Consider a linear page table for a process with only 1 page mapped to physical memory (paged in).

Number of valid page table entries (PTEs): 1 valid PTE

Total size of the page table: $2^{20} \cdot 4 \text{ B} = 4 \text{ MiB}$

- 4.3 Consider a 2-level page table for a process with only 1 page mapped to physical memory (paged in). Assume that VPN bits are split equally between the two levels.

Number of valid page table entries (PTEs): 2 valid PTEs

Total size of the page table structure:

- Level-1 page table = 2^{10} PTEs $\cdot 4 \text{ B} = 4 \text{ KiB}$
- Level-2 page table = 2^{10} PTEs $\cdot 4 \text{ B} = 4 \text{ KiB}$
- **8 KiB total**

5 Hierarchical Page Table Example

Assume: 8-bit virtual addresses, 32-bit words, 32-bit PTEs, 16-byte pages, a two-level page table (VPN bits split equally between levels), and an LRU 4-entry TLB.

At the beginning, the TLB is empty and the free pages list contains 0x9, 0x5, 0xA, 0x7, 0x1, 0x3, 0xB, 0xD, 0xE, and 0xF in that order. PTBR is set to 0.

5.1 How many bytes of virtual memory are addressable?

$2^8 = 256$ bytes

5.2 How many bytes of physical memory are addressable?

PPN = 32 bits; 2^{32} pages $\cdot$ 16 B/page = 64 GiB

5.3 Why might DRAM size $>$ virtual address space size be useful?

Multiple processes resident in main memory.

5.4 Divide the virtual address into:

Offset bits: `vaddr[3:0]`

Second-level index bits: `vaddr[5:4]`

Top-level index bits: `vaddr[7:6]`

5.5 How many entries are in the:

Top-level table? 4

Second-level tables (each)? 4

6 Page Table Walk Example

Same setup as above. Fill in the table for the following sequence of accesses, updating the page table in memory, the TLB, and the free pages list as you go.

6.1

Virtual Address	Index1	Index2	TLB hit/miss	Page hit / page fault	Physical Address
0x68	0x1	0x2	miss	hit	0x128
0x14	0x0	0x1	miss	hit	0x134
0x6C	0x1	0x2	hit	hit	0x12C
0x90	0x2	0x1	miss	fault	0x090
0x74	0x1	0x3	miss	hit	0x114
0xE4	0x3	0x2	miss	fault	0x0A4
0x18	0x0	0x1	miss	hit	0x138
0xD0	0x3	0x1	miss	fault	0x070

TLB (final contents)

VPN	0x7	0xE	0x1	0xD
PPN	0x11	0xA	0x13	0x07

Free pages: 0x9, 0x5, 0xA, 0x7, 0x1, 0x3, 0xB, 0xD, 0xE, 0xF

Free pages (after): 0x1, 0x3, 0xB, 0xD, 0xE, 0xF

Addr	Contents
0x00	0x06
0x04	0x04
0x08	0x02
0x0C	0x05
0x10	
0x14	
0x18	
0x1C	
0x20	0x08
0x24	0x09
0x28	
0x2C	
0x30	
0x34	
0x38	
0x3C	
0x40	
0x44	
0x48	0x12
0x4C	0x11
0x50	
0x54	0x07
0x58	0x0A
0x5C	
0x60	
0x64	0x13
0x68	
0x6C	

Walkthrough. Index1 = $\text{vaddr}[7:6]$, Index2 = $\text{vaddr}[5:4]$, offset = $\text{vaddr}[3:0]$. The top-level PTE for Index1 = i is at physical address $4i$; it holds the page number of the second-level table, whose PTE for Index2 = j is at $(\text{page number} \cdot 16) + 4j$. Physical address = $(\text{PPN} \cdot 16) + \text{offset}$.

- (a) **0x68** = 0110 1000: Index1 = 1, Index2 = 2. 0x04 $\rightarrow$ 0x04, so the L2 table is at 0x40; entry 2 at 0x48 $\rightarrow$ 0x12. PA = 0x128. TLB miss; insert 0x6 $\rightarrow$ 0x12.
- (b) **0x14** = 0001 0100: Index1 = 0, Index2 = 1. 0x00 $\rightarrow$ 0x06, L2 table at 0x60; entry 1 at 0x64 $\rightarrow$ 0x13. PA = 0x134. TLB miss; insert 0x1 $\rightarrow$ 0x13.

- (c) **0x6C**: VPN 0x6 is in the TLB → hit. PA = 0x12C.
- (d) **0x90** = 1001 0000: Index1 = 2, Index2 = 1. 0x08 → 0x02, L2 table at 0x20; entry 1 at 0x24 is empty → **page fault**. Allocate free page 0x9 (0x24 → 0x09). PA = 0x090. Insert 0x9 → 0x9.
- (e) **0x74** = 0111 0100: Index1 = 1, Index2 = 3. L2 table at 0x40; entry 3 at 0x4C → 0x11. PA = 0x114. TLB miss; insert 0x7 → 0x11. TLB is now full: {0x6, 0x1, 0x9, 0x7}.
- (f) **0xE4** = 1110 0100: Index1 = 3, Index2 = 2. 0x0C is empty → **page fault**: allocate 0x5 for a new L2 table (0x0C → 0x05); entry 2 of that table at 0x58 is empty, so allocate 0xA (0x58 → 0x0A). PA = 0x0A4. Insert 0xE → 0xA, evicting LRU entry 0x1 (last used on access 2).
- (g) **0x18** = 0001 1000: VPN 0x1 was just evicted → TLB miss. Walk: 0x64 → 0x13, PA = 0x138. Re-insert 0x1 → 0x13, evicting LRU entry 0x6 (last used on access 3).
- (h) **0xD0** = 1101 0000: Index1 = 3, Index2 = 1. 0x0C → 0x05, L2 table at 0x50; entry 1 at 0x54 is empty → **page fault**. Allocate 0x7 (0x54 → 0x07). PA = 0x070. Insert 0xD → 0x07, evicting LRU entry 0x9.

Final TLB: {0x7 → 0x11, 0xE → 0xA, 0x1 → 0x13, 0xD → 0x07}. Pages 0x9, 0x5, 0xA, 0x7 were consumed from the free list, leaving 0x1, 0x3, 0xB, 0xD, 0xE, 0xF.