CS152 Discussion Section 4

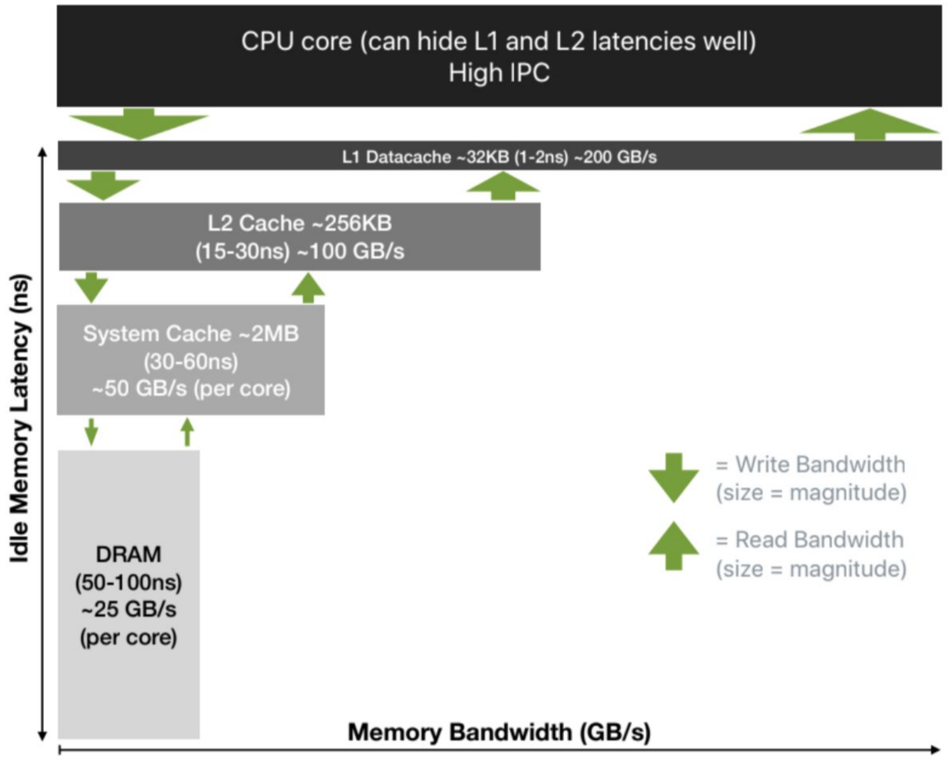
Virtual Memory + Lab 2

Week of Feb 12
Spring 2024

Agenda

- Prefetching
- Virtual Addresses
- Page Tables
- Lab 2 Preview

Memory Hierarchy Revisited (Better Picture)



Data is most likely to be in DRAM before first access. Long latencies to access DRAM

- e.g. 100ns -> 400 CPU cycles per access (at 4Ghz). This is ***difficult to hide***.
- On the other hand, L1 cache latency is 1ns -> 4 CPU cycles per access. ***Easier to hide***

Prefetching Metrics

- Accuracy
- Coverage
- Timeliness

Prefetching Metrics

- Accuracy
  - Is the prefetch useful (did we use what was prefetched)?
  - Useful / Total prefetches
- Coverage
  - Is the prefetcher covering all accesses?
  - Useful / Total unique accesses
- Timeliness
  - Is the prefetch on time (not too early / too late)?
  - On-time / Total prefetches

Prefetching Types

- Instruction Prefetching
  - What is the memory access pattern?
- Data Prefetching
  - What is the memory access pattern?

Prefetching Types

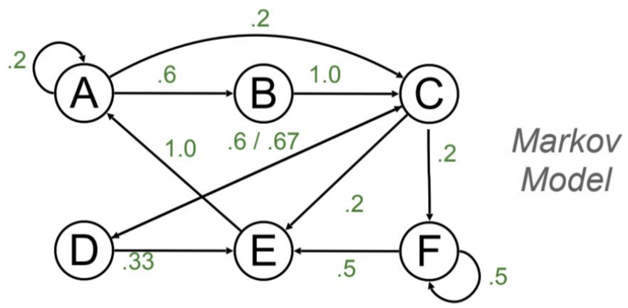
- Instruction Prefetching
    - What is the memory access pattern?
        - **Often sequential with control flow jumps**
- Data Prefetching
    - What is the memory access pattern?
        - **Much more irregular (loops, pointer chasing, etc)**

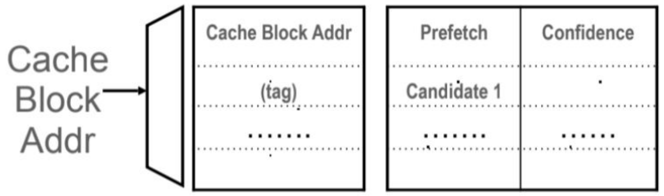
Prefetching Algorithms

- Next-Line
  - Always prefetch the next N cache lines after a demand access/compulsory miss
- Strided
  - After seeing N addresses with distance of D between them, prefetch the current address plus the D offset (C + D)
  - Is Next-Line considered Strided?
- More complicated
  - History-based prediction: i.e. Address Correlation

Address Correlation Prefetching

- After training, you know the probability of one address followed by another
- Use the prediction for the next prefetch request





What's the similarities between this and branch prediction? Can we use multiple addresses to determine next prefetch?

Q1: Prefetching

```
int A[N][M]; // N=32, M=32
int sum = 0;
for (int j = 0; j < M; j++) {
    for (int i = 0; i < N; i++) {
        prefetch(&A[i][j] + OFFSET); // prefetches from (A + M*i + j + OFFSET)
        sum += A[i][j];
    }
}
```

Assume 128B cache lines (each row fits entirely in a cache line). Without the prefetch, the inner loop takes 50 cycles. The L1 miss penalty is 40 cycles. What should OFFSET be to minimize the total program cycles?

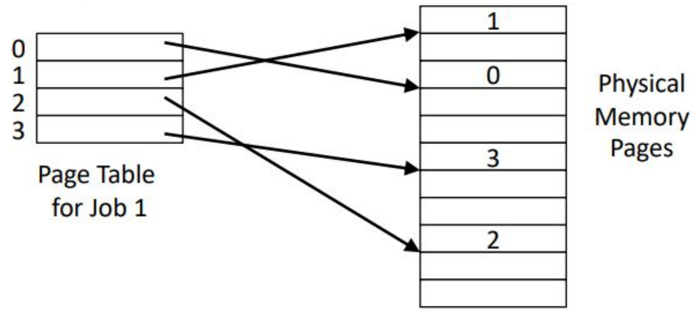
Q1: Prefetching

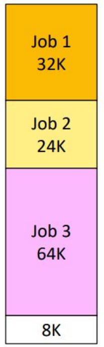
```
int A[N][M]; // N=32, M=32
int sum = 0;
for (int j = 0; j < M; j++) {
     for (int i = 0; i < N; i++) {
          prefetch(&A[i][j] + OFFSET); // prefetches from (A + M*i + j + OFFSET)
          sum += A[i][j];
     }
}
```

Perfect prefetching: 50-40=10 cycles per iteration. Prefetched data takes 40 cycles to return, so need to fetch 4 iterations in advance. OFFSET = 4*32 = 128

Virtual Memory: Page vs Segments

	Page	Segment
Replace a block	Easy (fixed size)	Difficult (variable size, hard to find in main memory)
Inefficiency	Internal	External
Efficiency in disk traffic	Yes (adjust page size to balance access time and transfer time)	Not always (bad when the segment is small)



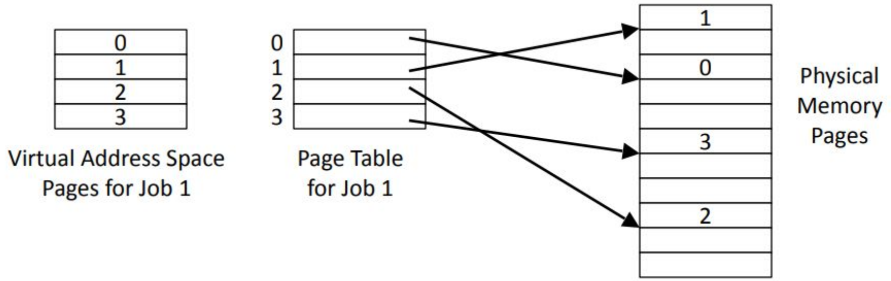


Linear vs Hierarchical Page Tables

- Program-generated (*virtual* or *logical*) address split into:

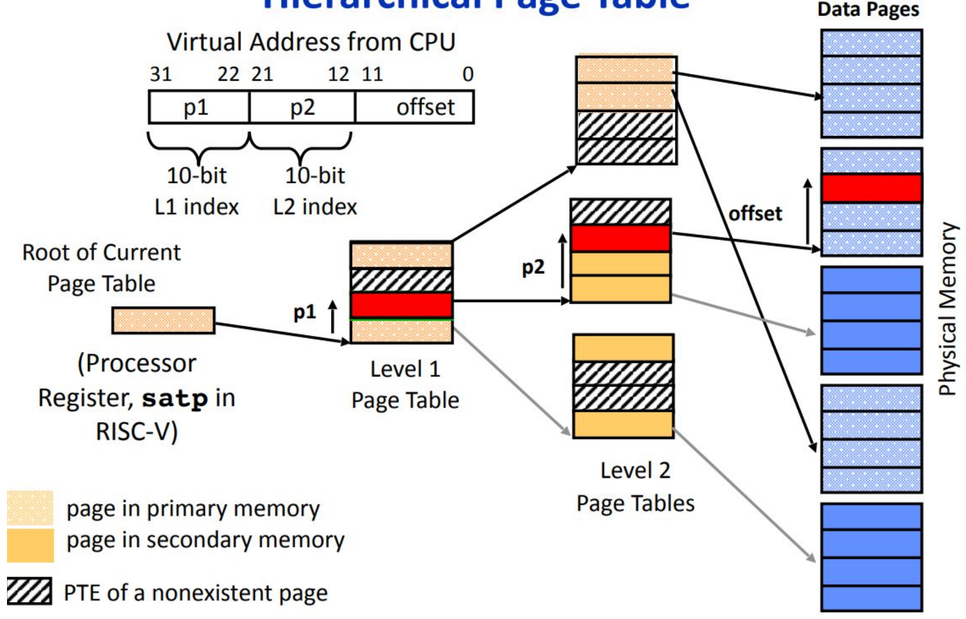
Page Number	Offset

- Page Table contains physical address of start of each fixed-sized page in virtual address space



- Paging makes it possible to store a large contiguous virtual memory space using non-contiguous physical memory pages

Hierarchical Page Table



RISC-V Sv32 Virtual Memory Scheme

Q2: Linear vs Hierarchical Page Tables

Consider 4 GiB (32-bit) of addressable virtual memory, 4 KiB pages, 4-byte PTEs (PPN, valid)

- Vaddr: [Virtual page number] [offset]
- How many bits in the page offset?
- How many bits in the page number?
- How many pages?

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

- How many valid PTEs?
- Total size of page table?

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.

- How many valid PTEs?
- Total size of page table?

Q2: Linear vs Hierarchical Page Tables

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

- How many bits in the page offset? log2(page size in bytes) = log2(4096) = 12
- How many bits in the virtual page number? Size of memory address - page offset bits = 32 - 12 = 20
- How many virtual pages? 2^20 PTEs

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

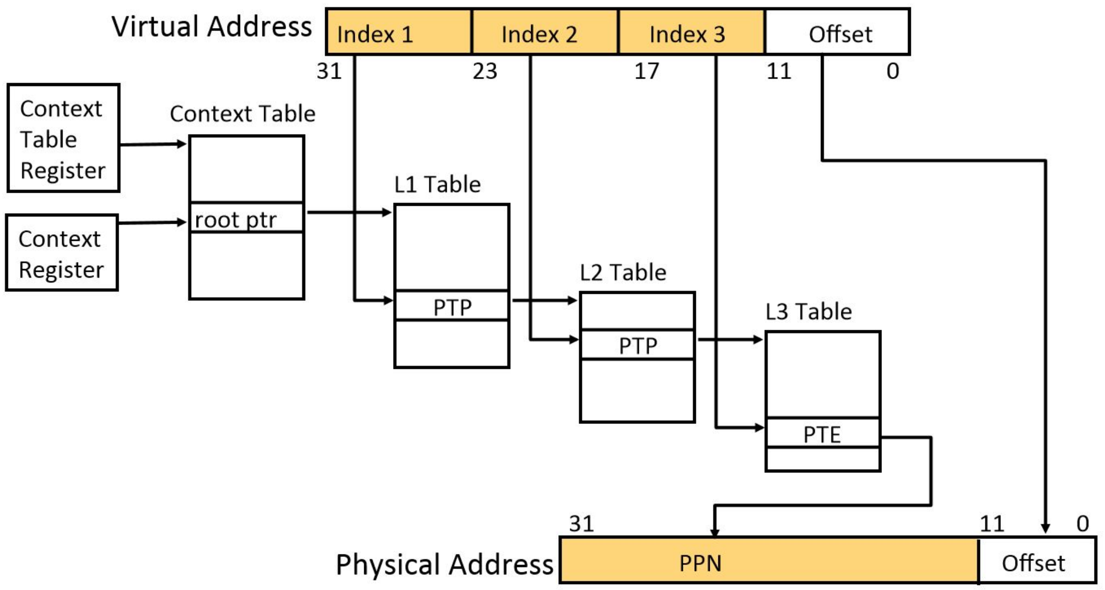
- How many valid PTEs? 1 valid PTE
- Total size of page table? 2^20 * 4B = 4MiB

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.

- How many valid PTEs? 2 valid PTEs
- Total size of page table structures?
  - Lvl 1 Page Table = 2^10 PTEs * 4 = 4KiB
  - Lvl 2 Page Table = 2^10 PTEs * 4 = 4KiB
  - 8 KiB total

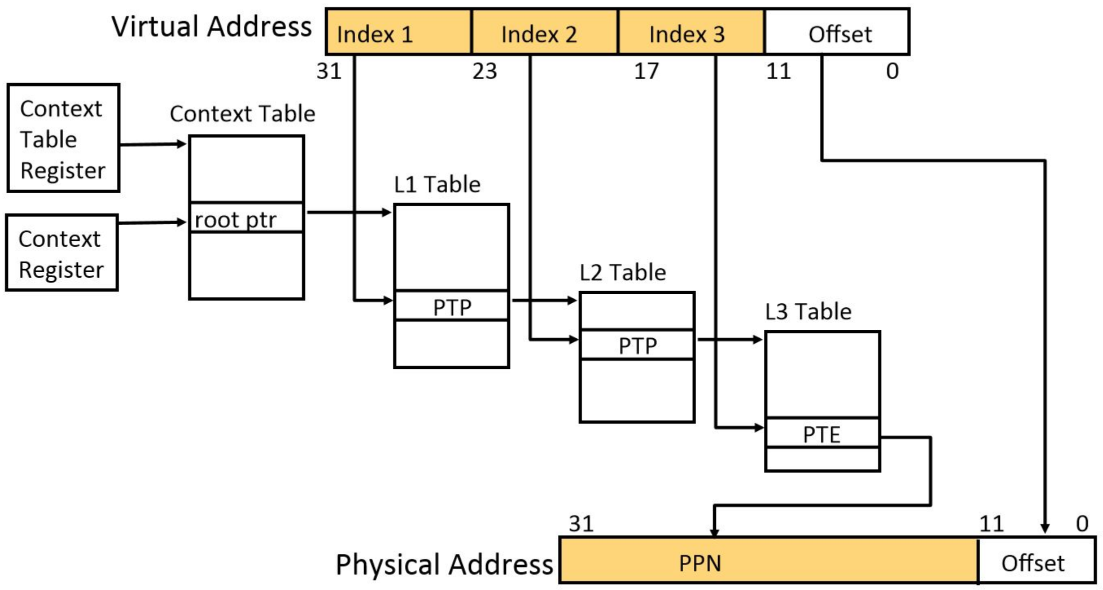
Memory Hierarchy with Virtual Memory

- **Page Table Walk**
  - Expensive
  - Software/Hardware
- Are virtual and physical addresses necessarily the same width?
- Can an architecture support multiple page sizes simultaneously?
  - Advantages and disadvantages of superpages?



Memory Hierarchy with Virtual Memory

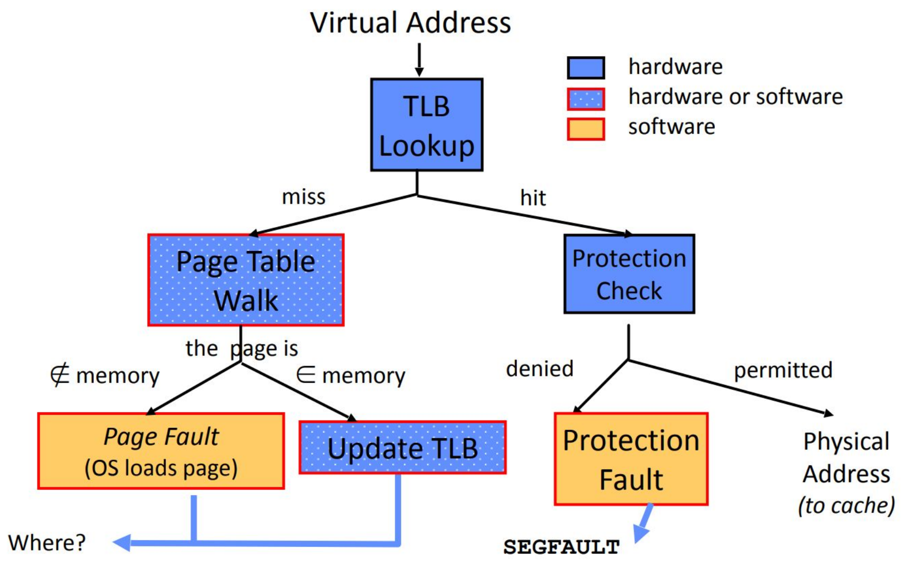
- **Page Table Walk**
  - Expensive
  - Software/Hardware
- Are virtual and physical addresses necessarily the same width? No
- Can an architecture support multiple page sizes simultaneously? Yes
  - Advantages and disadvantages of superpages?
    - Page faults penalties?
    - Paging traffic?
    - Internal fragmentation?



Translation Lookaside Buffer (TLB)

- **TLB**
  - Speed up address translation by caching PTEs
  - Typically fully associative
  - TLB miss is distinct from a page fault!

- On **ALL** accesses to a virtual address
  - Check for tag match in TLB
  - If no match:
    - Perform PTW
    - If no valid PTE (not paged in), page fault
  - Check protection bits:
    - If fail, page fault

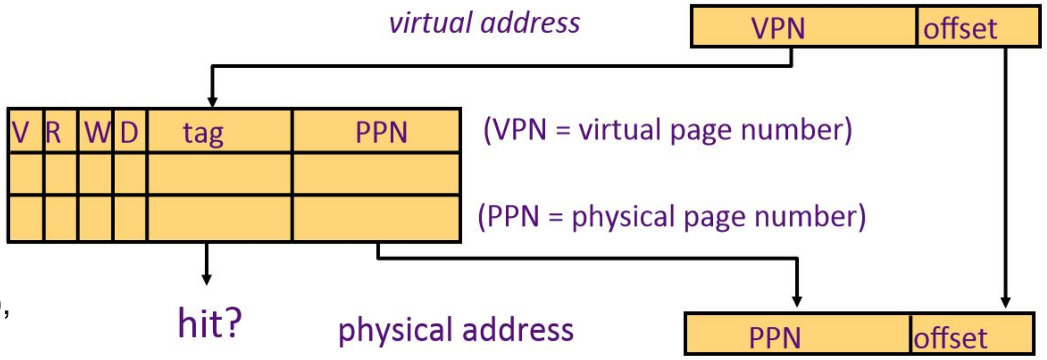


Translation Lookaside Buffer (TLB)

- **TLB**
  - Speed up address translation by caching PTEs
  - Typically fully associative
  - TLB miss is distinct from a page fault!

- On **ALL** accesses to a virtual address
  - Check for tag match in TLB
  - If no match:
    - Perform PTW
    - If no valid PTE (not paged in), page fault
  - Check protection bits:
    - If fail, page fault

Refer to online **Appendix L** of textbook for exhaustive treatment on TLB design



Multi-Level Page Tables

- What are the advantages of a hierarchical page table?
    - Disadvantages?
- What’s in a PTP? (Page table pointer)
- What’s in a PTE? (Page table entry)
- What manages the page tables?
- Should page tables reside in cache? Where?
    - When should PTEs enter the TLB?
    - Should page tables reside in data cache? What are the advantages/disadvantages?
- What if a PTE is modified in the page table while currently present in the TLB?
    - What if that PTE is in multiple TLBs (i.e., different cores)?

Multi-Level Page Tables

- What are the advantages of a hierarchical page table? reduce total page table size
  - Disadvantages? Longer page table walks
- What's in a PTP? (Page table pointer) pointer to index (PPN) of next level page table
- What's in a PTE? (Page table entry) PPN + valid bit, etc.
- What manages the page tables? OS (SW)
- Should page tables reside in cache? Where? Yes (SW uses standard memory instructions to manage page tables, e.g. paging in from main memory) Note that page table walks may not populate the cache if done with hardware.
  - When should PTEs enter the TLB? on each PT walk
  - Should page tables reside in data cache? What are the advantages/disadvantages?
    - PT walk -> likely you will walk down nearby PTs again
    - Potentially polluting cache
- What if a PTE is modified in the page table while currently present in the TLB?
  - If OS modifies PTE, need to evict that TLB entry
  - What if that PTE is in multiple TLBs (i.e., different cores)?
    - May need flush to all other TLBs (TLB shootdown)

Page Faults

Page faults represent a variety of causes:

- Pages that were swapped out to secondary storage (disk)
- Pages that are part of the legitimate address space but not yet committed
- Copy-on-write with shared pages (e.g. forks) or zero-filled pages
- Emulating accessed/dirty bits in PTEs without hardware support
- Segfault

Most page faults that occur are not errors!

Page faults must be ***restartable*** exceptions

Q3: Multi-Level Page Tables

- Assume: 8-bit virtual addresses, 32-bit words, 32-bit PTEs, 16-byte pages, two-level page table, 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.

1. How many bytes of virtual memory are addressable?
2. How many bytes of physical memory are addressable? Assume a PTE is PPN + valid bit
3. Why might DRAM size > virtual address space size be useful?

Q3: Multi-Level Page Tables

- Assume: 8-bit virtual addresses, 32-bit words, 32-bit PTEs, 16-byte pages, two-level page table, 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.

Physical address = [PPN] [4 bits]

1. How many bytes of virtual memory are addressable?
    a. 2^8 = 256 Bytes
2. How many bytes of physical memory are addressable? Assume a PTE is PPN + valid bit
    a. PPN = 32 - 1 = 31 bits; 2^31 pages * 16 B/page = 32 GiB
3. Why might DRAM size > virtual address space size be useful?
    a. Multiple processes resident in main memory

Q3: Multi-Level Page Tables

- 8-bit virtual addresses,
- 32-bit words,
- 16-byte pages,
- two-level page table,
- LRU 4-entry TLB

Offset bits: log2(page size) = 4

VPN bits: VA width - offset bits = 8-4 =4

Index1 bits: 2

Index2 bits: 2

offset (vaddr[a:b]) : vaddr[3:0]

index2 (vaddr[a:b]) : vaddr[5:4]

index1 (vaddr[a:b]) : vaddr[7:6]

Q3: Multi-Level Page Tables

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

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					
0x74					
0xE4					
0x18					
0xD0					

TLB				
VPN	0x6	0x1		
PPN	0x12	0x13		

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

Q3: Multi-Level Page Tables

Free pages: 0x9, 0x5, 0xA, 0x7, 0x1, 0x3 (PPNs)

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				
VPN	~~0x6~~ 0x1	~~0x1~~ 0xe	~~0x9~~ 0xd	0x7
PPN	~~0x12~~ 0x13	~~0x13~~ 0xA	~~0x9~~ 0x07	0x11

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	

Lab 2

Focuses on design of memory hierarchies using realistic SoC implementations

- Directed problem: Matrix transpose case study
- Open-ended problems:
    a. Reverse-engineer a memory system using micro-benchmarks
    b. Design your own hardware prefetcher (recommended to run on eda-* machines)
    c. Design your own replacement policy and victim cache

Fair Warnings

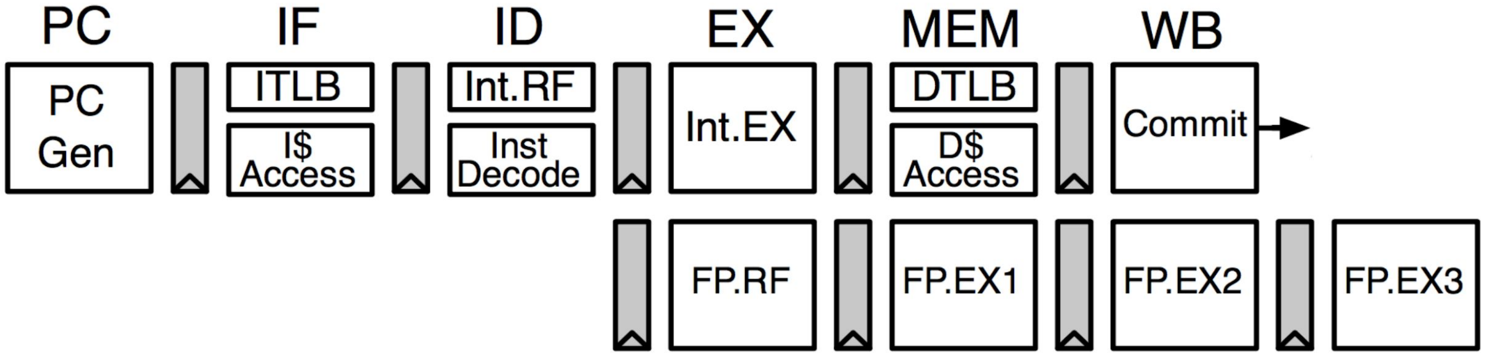
Expect to spend a significant fraction of time on RTL simulations

- Some interesting memory behaviors only manifest over longer time scales
- ~4.5 kHz simulator (varies by design complexity)
- 0.5-3 million cycles for a “short” program (2 to 10 minutes)
  - Up to 10 million cycles for a few benchmarks (bfs, ccbench)
  - Potentially long debug cycle for some open-ended problems
- Can run parallel simulations in some cases (`make -j` *`N`*)

Budget your time appropriately - start early!

- Option to not use EDA machines (and/or use multiple ones)

Rocket



- Single-issue in-order RV64IMAFDC core
- Extensively optimized for efficient ASIC implementation (1.6 GHz in 28nm)
    - Minimize high-fanout stall signals
    - Restructure pipeline logic to cope with long clock-to-Q delays of compiler-generated SRAMs
    - Details in background section of Lab 2 document
- Implements privileged ISA