18-344 Recitation 7

10/31/2025

Outline

1. Logistics
2. Review
3. Lab 3

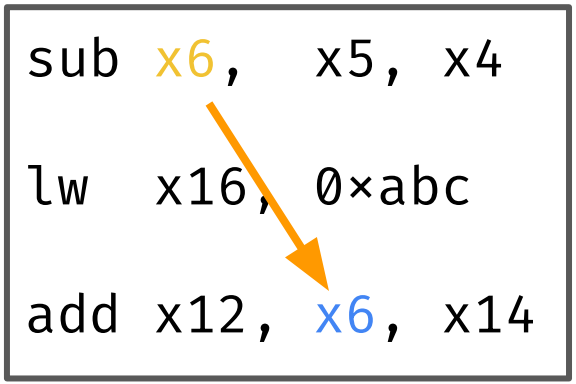
Logistics

- Homework 7 released, due November 5 (Wednesday) (in 5 days)
  - Covers lectures 15-16

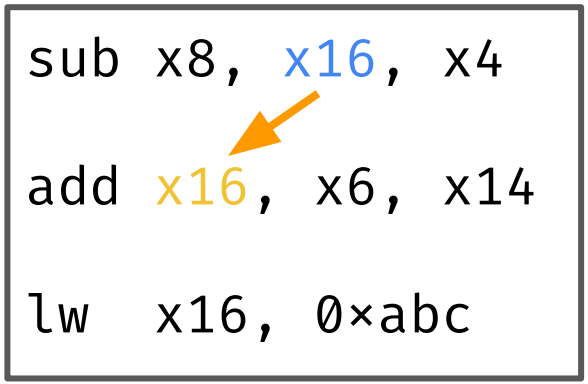
- Lab 3
  - Code freeze: October 14 (Friday) (in two weeks)
    - Submit (push a commit to your repo) on GitHub Classroom
  - Report: November 17 (Monday) (in two weeks + three days)
    - Submit on Gradescope

Review

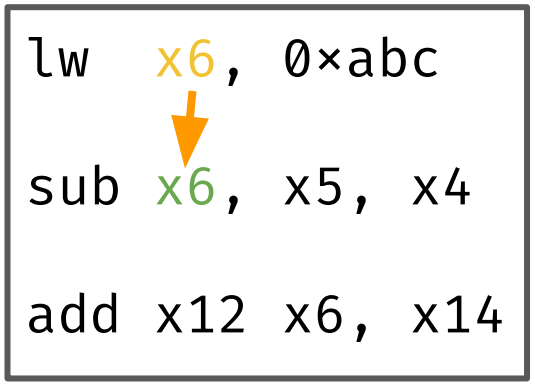
Pipeline ***data*** hazards



Read-After-Write (RAW)



Write-After-Read (WAR)



Write-After-Write (WAW)

There's always data hazards in programs, and we can't change that.

How to further exploit ILP?

Exploiting ILP using multiple issue

Common name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
Superscalar (static)	Dynamic	Hardware	Static	In-order execution	Mostly in the embedded space: MIPS and ARM, including the Cortex-A53
Superscalar (dynamic)	Dynamic	Hardware	Dynamic	Some out-of-order execution, but no speculation	None at the present
Superscalar (speculative)	Dynamic	Hardware	Dynamic with speculation	Out-of-order execution with speculation	Intel Core i3, i5, i7; AMD Phenom; IBM Power 7
VLIW/LIW	Static	Primarily software	Static	All hazards determined and indicated by compiler	Most examples are in signal processing, such as the TI
EPIC	Primarily static				

What we mean when we say “superscalar out-of-order” *in this class*.

Computer Architecture: A Quantitative Approach 6th Edition, Hennesy and Patterson

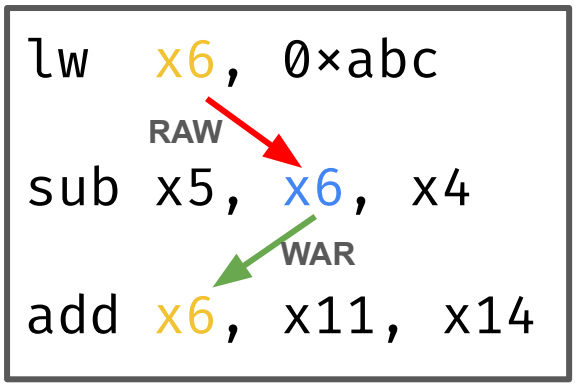
Exploiting ILP using multiple issue

Hardware-based speculation:

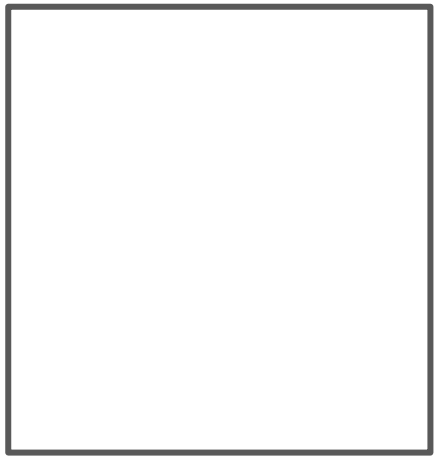
1. Dynamic branch prediction to choose which instructions to execute
2. Speculation to allow the execution of instructions before the control dependences are resolved
3. Dynamic scheduling to deal with the scheduling of different combinations of basic blocks

Common	Issue	Hazard		Distinguishing	
				speculation	
Superscalar (speculative)	Dynamic	Hardware	Dynamic with speculation	Out-of-order execution with speculation	Intel Core i3, i5, i7; AMD Phenom; IBM Power 7
VLIW/LIW	Static	Primarily software	Static	All hazards determined and indicated by compiler (often implicitly)	Most examples are in signal processing, such as the TI C6x
EPIC	Primarily static	Primarily software	Mostly static	All hazards determined and indicated explicitly by the compiler	Itanium

Register renaming



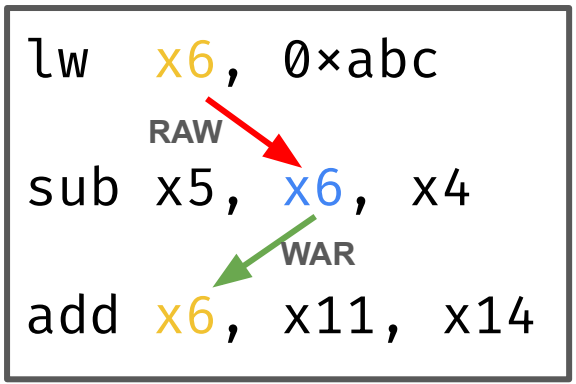
Instruction stream



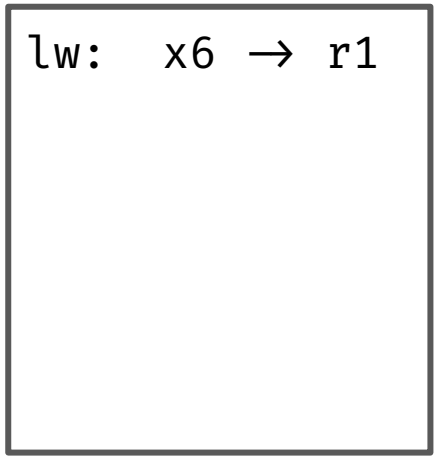
Renaming table

Instruction stream after renaming

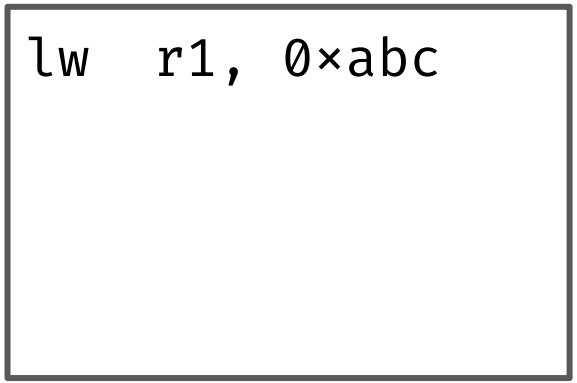
Register renaming



Instruction stream

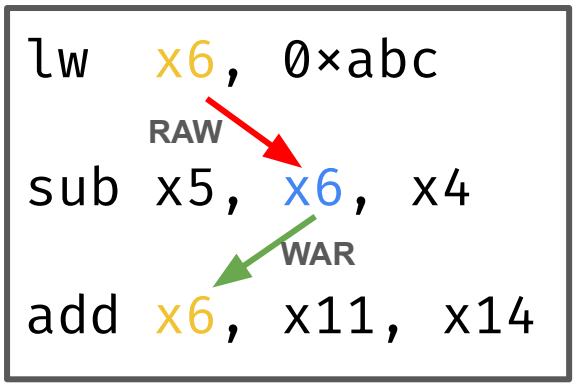


Renaming table

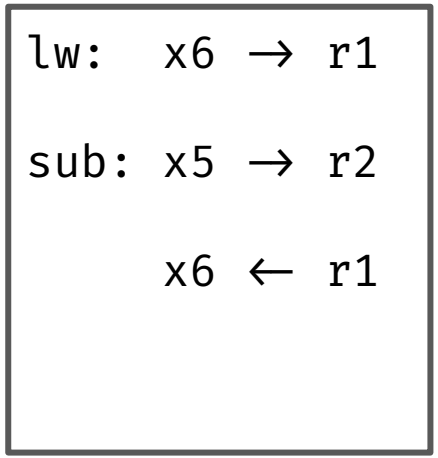


Instruction stream after renaming

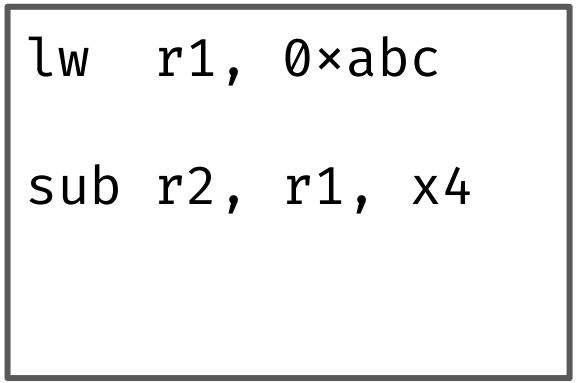
Register renaming



Instruction stream

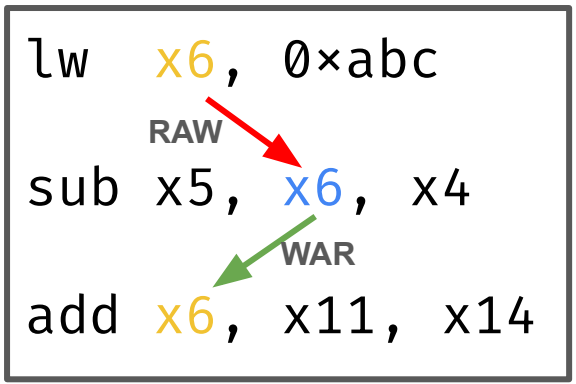


Renaming table

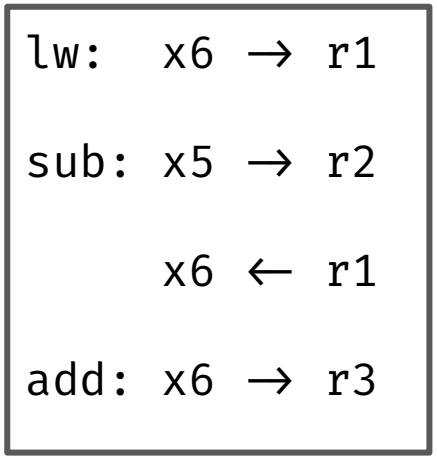


Instruction stream after renaming

Register renaming



Instruction stream

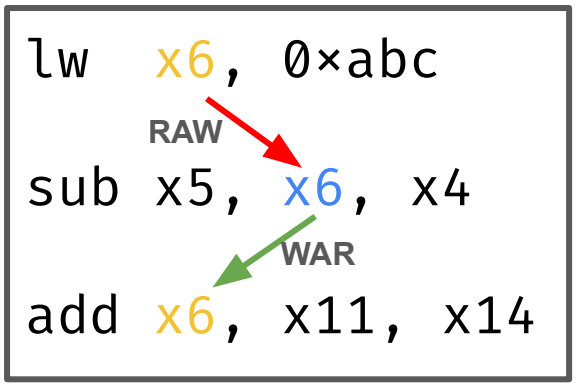


Renaming table

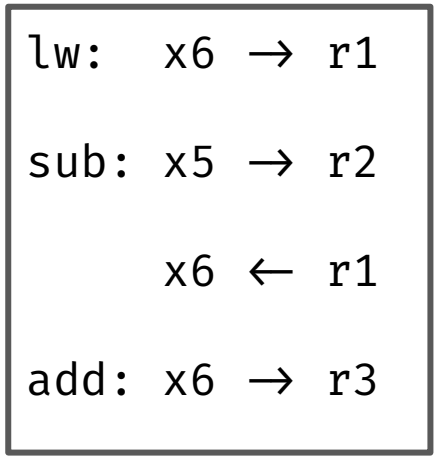
```
lw  r1, 0×abc

sub r2, r1, x4

add r3, x11, x14
```

Instruction stream after renaming

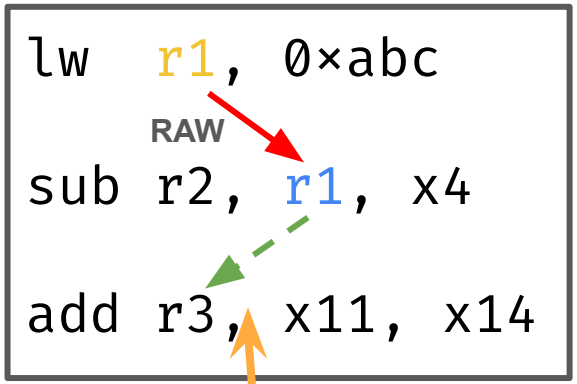
Register renaming



Instruction stream



Renaming table



Instruction stream after renaming

Gets rid of the false dependence.

Lab 3

Overview

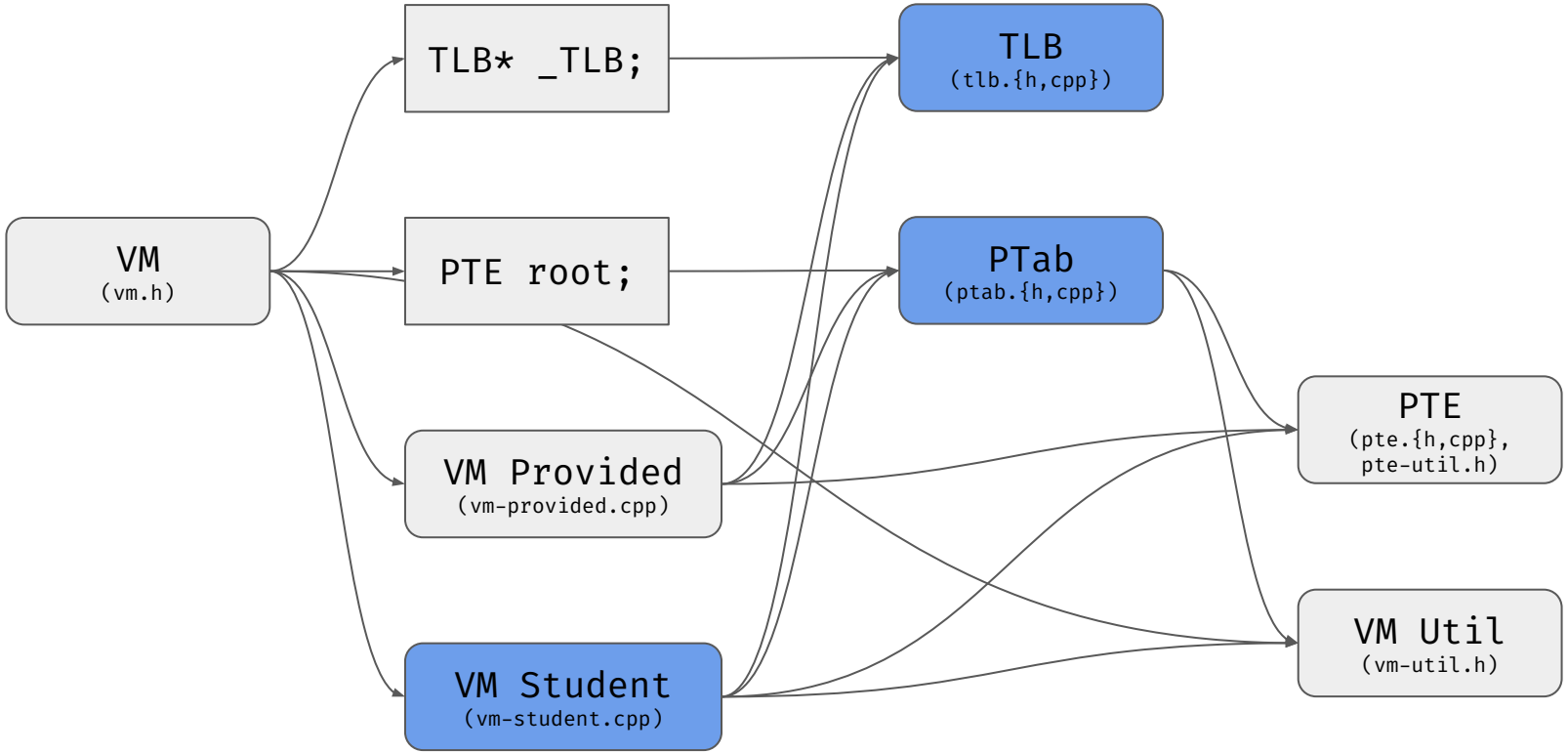
- A VM simulator, instead of a pintool
  - Virtual-to-physical mapping using hierarchical page table
  - Translation Lookaside Buffer
  - Page fault handler

What's provided, and what's not

- Provided by us, can make small changes if you want
  - `pte-util.h, pte.{h,cpp}`
    - Simple declarations of PTE types for you to use
  - `vm.h, vm-util.h`
    - VM class and various helpful declarations for you to use
  - `Vm-provided.cpp`
    - Contains implementation of VM methods that you do not need to implement (e.g. allocating and replacing physical pages)
  - `vm_trace/*`
    - A pintool we provide that you can use to collect a trace of memory accesses for testing

- Provided, but you'll need to implement/modify
  - `ptab.{h,cpp}`
    - Skeleton code and declarations of page table operations that you'll need to implement
  - `tlb.{h,cpp}`
    - Skeleton code and declarations of a TLB that you'll need to implement
  - `vm-student.cpp`
    - This is where you'll implement VM methods (e.g. map, translate) using the page table and TLB
  - `vm-test.cpp`
    - A driver for the VM simulator that runs on a given trace. Skeleton code with a simple, hard-coded trace is provided. You'll need to modify it so it works with traces generated by `vm_trace`.

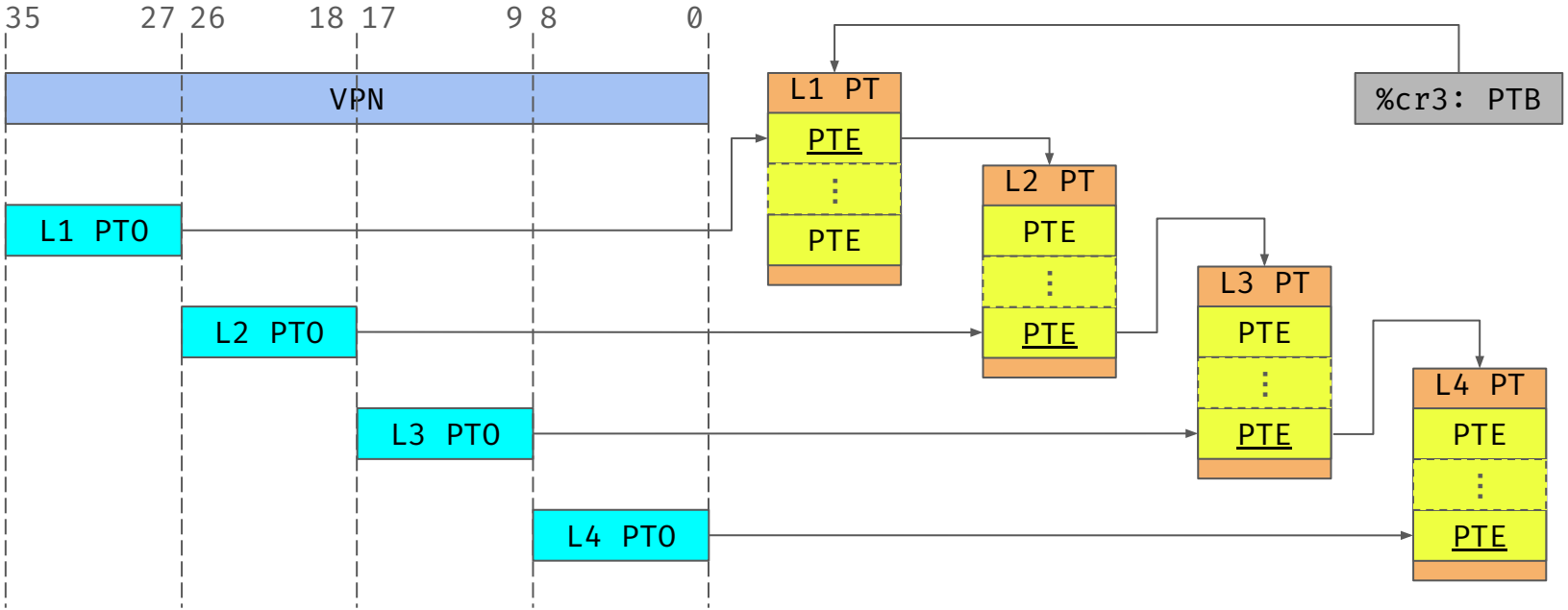
General structure



VPN: Virtual Page Number
PT: Page Table
PTB: Page Table Base
PTE: Page Table Entry
PTO: Page Table Offset

Task 1: Implement a hierarchical page table

48-bit virtual addresses and 4KB pages => 512 entries per page table.



Task 1: Implement a hierarchical page table

`vm-student.cpp`: implement three functions (using your PT implementation):

- `vmMap(vaddr, size)`
  - Update Page Table to map `size` bytes at address `vaddr`; Could span multiple pages; create Page Tables or PTEs here if not exist
- `vmTranslate(vaddr)`
  - TODOs in the handout; Return the translated physical address
- `vmPageFaultHandler(*pte)`
  - Handle Page Faults (page not residing in memory) here; if there exist free physical pages in memory, allocate them with `bumpAllocate()` function; if you run out of free pages, replace an existing page with the `replacePage()` function. These functions return the PPN to store in your PTE.
  - `bumpAllocate()` and `replacePage()` are provided to you; declarations in `vm.h`, implementations in `vm-provided.cpp`.

Task 2: Implement a TLB

`tlb.{h,cpp}`: Implement a TLB structure, with your choice of organization (size, ways, replacement). You will also need two functions: `lookup()` and `update()`.

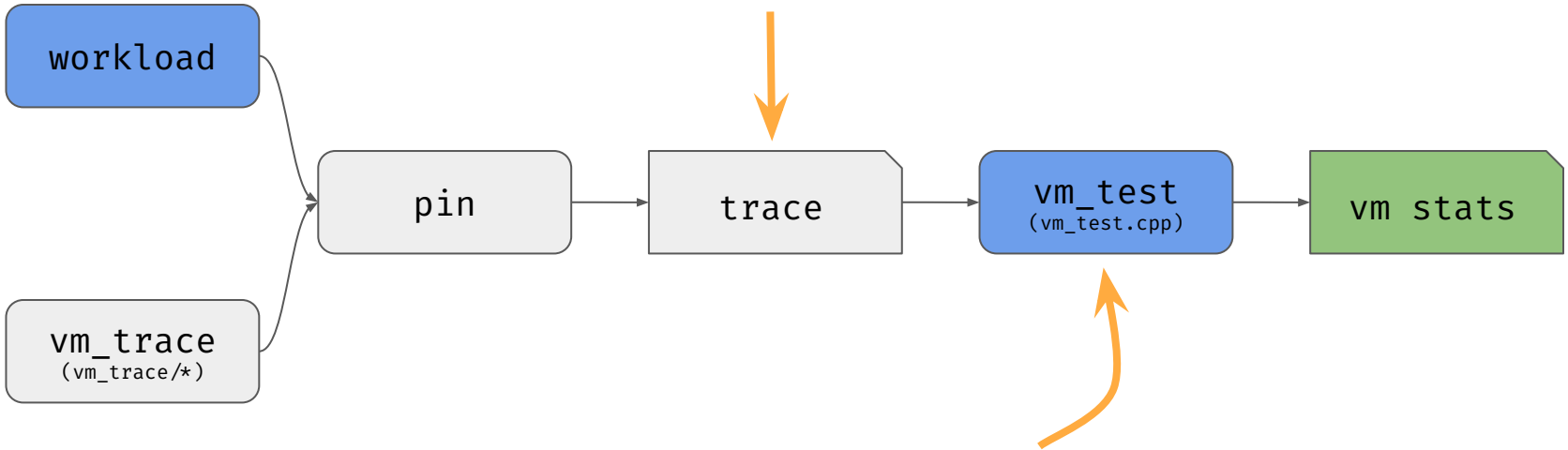
- `lookup(vaddr, &PPN)`
  - Attempt to assign cached-`PPN` to the argument `&PPN`, and return true for hits.

- `update(vaddr, new_PPN)`
  - Update the TLB entries with this new mapping.
  - This may evict an existing entry, your TLB organization should account for replacement.

Testing

Sample `mmap` log:

`PC= ... : mmap(vaddr = ... len = ... <zip> ) ret = ...`



Modify `vm_test.cpp` to parse* the traces generated by `vm_trace`.
`vm_trace` logs `mmaps` and optionally memory accesses.

Reporting the results

- Page Table implementation should report `#page_faults`, `#num_accesses` and `#tlb_hits`.

- You will justify your TLB organization with a ***quantitative*** analysis of these three parameters.
  - Show a plot of TLB misses vs TLB configurations.
  - Reason about the reduction in Page Table walks due to your TLB.

- Your code should also be robust to ***handle exceptions***: page faults and segmentation faults (unmapped accesses).

- Turn in all code, writeup, and test traces!
  - Include any scripts/tests you used as well.