Project 1
Infrastructure

- L1 receives at most one load/store per cycle
- L1 has priority over prefetcher
- L2 is fully pipelined
  - Max 10 outstanding requests
  - 20 cycle latency
- DRAM is not pipelined
  - 50 cycle latency
Prefetcher Interface

- **bool hasRequest(u_int32_t cycle)**
  - Does your prefetcher want to issue a request now?
- **Request getRequest(u_int32_t cycle)**
  - Get a request from your prefetcher.
- **void CompleteRequest(u_int32_t cycle)**
  - The L2 accepted your request. This could occur many cycle after getRequest() was called, if the L2 queue is full.
- **void cpuRequest(Request req)**
  - The CPU issued a request to the L1. Here is what happened.
*Anatomy of a Request*

```c
struct Request {
    // effective address of request (32 bits). Set this in requests you generate. All other fields will be ignored.
    u_int32_t addr;
    // PC of request (32 bits)
    u_int32_t pc;
    // true for loads, false for store
    bool load;
    // Not for your use
    bool fromCPU;
    // cycle that the request was issued
    u_int32_t issuedAt;
    // if it was a hit in D-cache
    bool HitL1;
    // Not for your use.
    bool HitL2;
};
```
Example: prefetcher.h

```cpp
#include <sys/types.h>
#include "mem-sim.h"

class Prefetcher {
private:
    bool _ready;
    Request _nextReq;

public:
    Prefetcher();
    bool hasRequest(u_int32_t cycle);
    Request getRequest(u_int32_t cycle);
    void completeRequest(u_int32_t cycle);
    void cpuRequest(Request req);
};
```

Prefetcher state

No need to mess with this.
Example: prefetcher.C

```c
#include "prefetcher.h"
#include <stdio.h>

Prefetcher::Prefetcher() { _ready = false; }

bool Prefetcher::hasRequest(u_int32_t cycle) { return _ready; }

Request Prefetcher::getRequest(u_int32_t cycle) { return _nextReq; }

void Prefetcher::completeRequest(u_int32_t cycle) { _ready = false; }

void Prefetcher::cpuRequest(Request req) {
    if(!_ready && !req.HitL1) {
        _nextReq.addr = req.addr + 16;
        _ready = true;
    }
}
```
Build and run

>> cd cse240a-proj1/
>> make
>> ./cacheSim testgen.trace

Creates testgen.trace.out
Output

testgen.trace.out:
830778    Total run time (cycles)
0.9575    Dcache total hit rate
0.5363    L2 total hit rate
2.8362    Average memory access time
0.2142    Average L2 queue length
0.0092    D-cache - L2 bw utilization
0.2142    Memory utilization bandwidth

You may want to hack infrastructure to collect additional statistics.
Deliverables

• Written description of your prefetcher
  – How much state? Limit of 4KB
  – AMAT graph for the traces on the web site
  – Citations, if needed.
  – lastname-firstname-cse240a-sp14-prefetch.pdf
  – Emailed to Skanda.

• Source code
  – prefetcher.c and prefetcher.h
  – They must build on the APE lap machines.
  – They work with the infrastructure source posted on the web site.

• Due May 27th.
Deliverables contd.

• You can work on the project in pairs. One person can submit the project but make sure to include both names on the report.
• For this project we will have a leaderboard that will be updated at the end of every week.
• 5% of the project grade is for submitting your latest fully functioning version of the prefetcher by the Friday of each week.
• Extra points for creative names for the prefetcher !!!
What to implement?

- Papers on the project web page
- Use (and make sure to cite) one of them or make something up.
- Go crazy. 4K is a lot of state for a prefetcher.
Contest

• Best AMAT wins

• Traces of the TAs and my choosing
  – Test your design thoroughly
  – Generate traces of your own (pin tool on the web page)
  – Share them if you like.

• Fame and fabulous prizes await.