Project 2
**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

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

#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

```bash
>> cd cse240a-proj2/
>> make
>> ./cacheSim testgen.trace

Creates testgen.trace.out
```
## Output

testgen.trace.out:

<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>830778</td>
<td>Total run time (cycles)</td>
</tr>
<tr>
<td>0.9575</td>
<td>Dcache total hit rate</td>
</tr>
<tr>
<td>0.5363</td>
<td>L2 total hit rate</td>
</tr>
<tr>
<td>2.8362</td>
<td>Average memory access time</td>
</tr>
<tr>
<td>0.2142</td>
<td>Average L2 queue length</td>
</tr>
<tr>
<td>0.0092</td>
<td>D-cache - L2 bw utilization</td>
</tr>
<tr>
<td>0.2142</td>
<td>Memory utilization bandwidth</td>
</tr>
</tbody>
</table>

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-wi10-prefetch.pdf
  – Emailed to Hung Wei

• 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.
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 Hung Wei 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.