HW6: MemPath

ShanghaiTech University
CS110 Computer Architecture I
Spring 2026

Abstract

This homework studies the shared instruction/data memory path of a simplified RV32I system. The public scaffold already provides program loading, untimed functional execution, trace storage, replay, and final reporting. Your task is to complete the memory-facing logic that turns executed `lw`/`sw` instructions into data requests, models the behavior of a shared cache whose organization is determined by s, E, and b, and returns correct per-request timing through the memory controller. The central separation of the assignment is fixed: untimed execution determines what requests happen, while replay through the shared cache and DRAM determines how long those requests take.

Contents

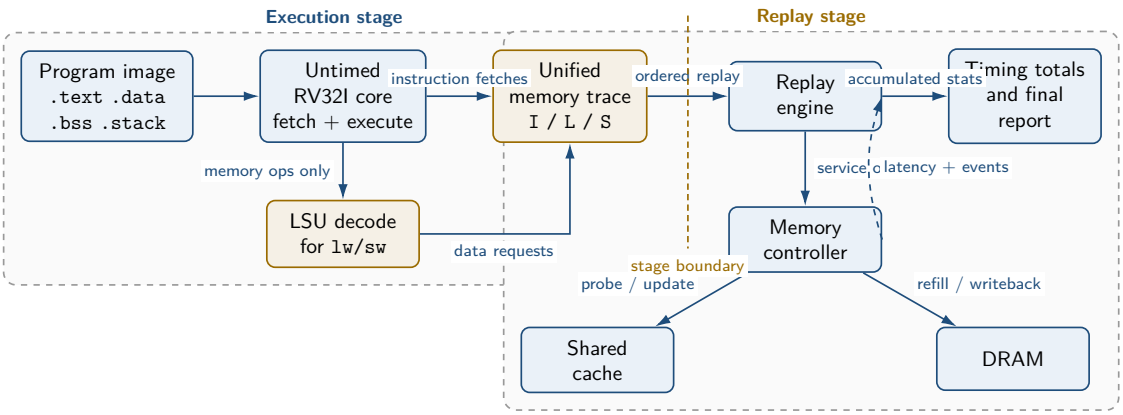

Figure 1: Execution-stage and replay-stage dataflow. The framework core and LSU determine the dynamic request stream, while replay drives the shared memory path and accumulates final timing statistics.

1 Introduction and Scope

The simulator in this repository executes a machine-code image and models one shared memory path for both instruction fetches and data accesses. Unlike a full CPU project, this homework does not ask you to build a pipeline, implement speculation, or redesign general instruction execution. The framework already provides an untimed in-order RV32I core precisely because raw machine code alone is not enough to reveal the dynamic request sequence: instruction fetch order depends on control flow, and data addresses depend on register values produced by earlier execution.

The assignment therefore has two tightly connected stages. During execution, the framework core and your LSU logic generate an ordered unified trace of `I`, `L`, and `S` requests. During replay, the simulator feeds that trace through one shared cache and one blocking DRAM model to obtain timing statistics. Preserving this split is essential: code on the execution side must not model timing, and code on the replay side must not rediscover instruction semantics. **Figure 1** summarizes this overall separation between untimed execution and timing replay.

The public scope is intentionally focused. The input program is RV32I machine code stored in a simple text image. Every executed instruction contributes an instruction-fetch request. Among data-memory instructions, only `lw` and `sw` are required. The memory system is blocking, deterministic, and single-level, with no separate I-cache, no separate D-cache, no virtual memory, and no speculative behavior. The goal is not restricted to the ordinary multi-way case: the same cache contract must continue to work for direct-mapped configurations ($E = 1$), one-set organizations ($s = 0$), and general multi-set, multi-way organizations.

2 Scaffold and Student Responsibilities

Table 1 summarizes the public scaffold and the student-owned implementation surface. When your code interacts with framework-owned modules, it should do so through the provided interfaces rather than by bypassing them or reimplementing their jobs locally. In particular, student code should not replace trace storage, DRAM timing, replay bookkeeping, or general

core behavior with parallel private mechanisms. Gradescope scoring uses only the required simulator output for correctness.

Table 1: Public scaffold ownership

Component	**Role in the simulator**	**Student work**
`src/image.c`	Loads the image and owns `.text`, `.data`, `.bss`, and `.stack`.	Read only.
`src/core.c`	Untimed RV32I execution, instruction fetch, and functional non-memory behavior.	Read only.
`src/trace/trace.c`	Ordered storage for unified `I/L/S` requests.	Read only.
`src/hw/dram.c`	Blocking DRAM timing model.	Read only.
`src/hw/lsu.c`	Memory-related instruction decode and data-request issuance.	Implement.
`src/hw/cache.c`	Shared cache lookup, state, replacement, and updates.	Implement.
`src/hw/mem_ctrl.c`	One-request cache/DRAM sequencing and latency accounting.	Implement or extend.
`src/sim/replay.c`, `src/sim/report.c`	Replay totals and final report formatting.	Read only.

2.1 Load/Store Unit

Your work in `src/hw/lsu.c` is responsible for real machine-code decode of `lw` and `sw`. Given the current instruction word and operand values supplied by the framework, the LSU must identify whether the instruction is a supported data-memory operation, extract the correct immediate, compute the effective byte address, determine the access size, append one `L` or `S` entry to the unified trace, and return a normalized request structure to the framework core. The LSU must not change the PC, update architectural registers directly, update functional memory directly, or model cache or DRAM timing. It must also use the provided trace interface rather than storing requests through an ad hoc side channel. The framework also enforces a consistency rule here: if the LSU returns `valid = false`, it must append no new trace entry; if it returns `valid = true`, it must append exactly one matching request whose `op`, `addr`, and `size` agree with the returned LSU result.

2.2 Shared Cache

Your work in `src/hw/cache.c` is responsible for the shared cache model. This includes deriving the correct cache location and identity information from an incoming byte address, together with hit detection, victim choice, LRU state, dirty state, and line installation. The cache API must support the full range of valid (s, E, b) organizations, including direct-mapped, one-set, and general multi-way cases. Deterministic behavior is required throughout: invalid lines must be chosen before valid evictions, and ties must be broken predictably. Replay and the controller should observe cache behavior only through the public cache interfaces; they must not reach into private cache state directly.

2.3 Memory Controller

Your work in `src/hw/mem_ctrl.c` is responsible for one logical request at a time. It should compute the per-request outcome for one logical request, coordinate cache and DRAM through the provided public APIs, keep the mandatory miss order, report only per-request hardware outcomes such as logical hit/miss and eviction/writeback events, and accumulate only the request-local `cache_cycles` and `dram_cycles` consumed while servicing that request. Replay uses the controller by passing each trace request into `mc_access()` and accumulating the returned per-request totals. For stores, it must preserve the fixed write-back, write-allocate policy. The required miss order is

$$\begin{aligned} \text{writeback (if dirty)} &\rightarrow \text{read incoming block} \\ &\rightarrow \text{install line} \\ &\rightarrow \text{mark dirty (if store)} \\ &\rightarrow \text{update LRU.} \end{aligned}$$

For one touched cache block, the controller should conceptually perform one cache lookup, decide hit or miss, write a dirty victim back if needed, read the incoming block if needed, install the new line, mark it dirty for a store when applicable, and then update LRU. This module is also the correct place to extend behavior for advanced split-line cases, because replay statistics are defined per logical request while the memory controller owns the detailed block-by-block service path. Inside `mem_ctrl.c`, timing behavior should be expressed through the provided interfaces rather than by short-circuiting the whole request into a precomputed answer. On the cache side, useful helpers include `cache_probe()`, `cache_hit_lat()`, `cache_on_hit()`, and `cache_on_fill()`. On the DRAM side, use `dram_read_lat()` and `dram_write_lat()` for the local DRAM timing contribution. If you want to reason about advanced split-line behavior, the scaffold also exposes helpers such as `cache_block_addr()` and `cache_block_base()`. Most of this assignment focuses on the regular one-block case, but that should not be read as a promise that multi-block requests are impossible. **Figure 2** shows the one-request service flow and the controller's input/output role in that path.

Everything outside these three files should be treated as framework support for this release. In particular, `src/core.c` owns general untimed RV32I execution, control flow, register-file state, untimed functional loads and stores after LSU decode, and stop conditions such as `ecall`. Those files are worth reading so you understand the simulator flow, but they are not the main implementation surface of the homework. Likewise, `replay.c` computes the final totals by calling `mc_access()` once per trace entry, while `mem_ctrl.c` owns the one-request service path itself.

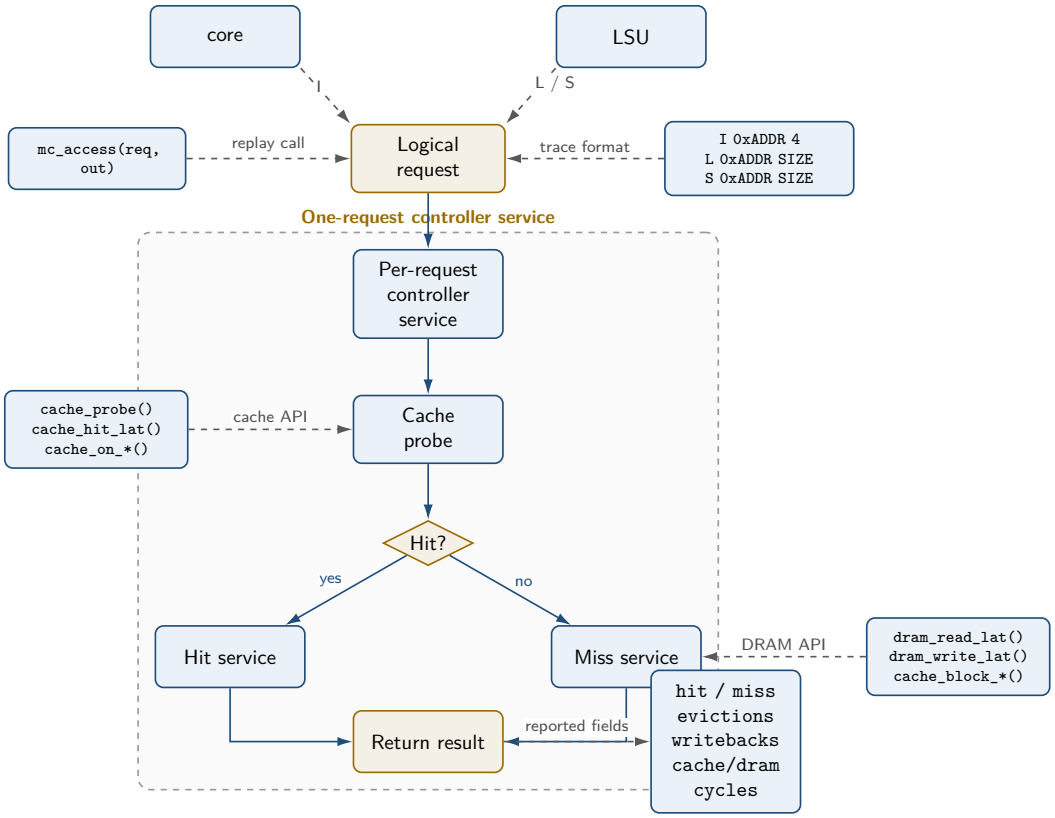

Figure 2: One-request service path in the memory controller. Replay issues one logical request through `mc_access()`, the controller services that request through the cache and DRAM helpers, and the completed per-request result returns to replay.

3 Input, Execution, and Trace Model

The simulator consumes a text image file so that test programs can be generated easily from small host C programs. The public image format is summarized in Table 2.

An example image is shown below:

```
entry 0x00001000
text 0x00001000 4
0x100000b7
0x0000a103
0x0020a223
0x00000073
data 0x10000000 2
0x11223344
0x00000000
bss 0x10001000 64
```

Numbers are parsed with base 0, so both decimal and hexadecimal forms are accepted.

The public executable interface is

Table 2: Image record format

Record	Meaning
`entry <addr>`	Initial PC value.
`text <base> <count>`	Base address and number of 32-bit instruction words that follow.
`data <base> <count>`	Base address and number of initialized 32-bit data words that follow.
`bss <base> <size_bytes>`	Zero-initialized writable region.
`stack <base> <size_bytes>`	Explicit stack segment; if omitted, the framework inserts a default stack at `0x7fff0000` of size `4096` bytes.

```
mempath_sim <image> <s> <E> <b>
            <hit_lat> <dram_read_lat> <dram_write_lat>
            <cpu_freq_ghz> <cache_freq_ghz> <dram_freq_ghz>
```

and the meaning of these arguments is summarized in Table 3.

Table 3: Command-line arguments

Argument	Meaning
`image`	Program image file.
`s`	$\log_2$ of the number of sets.
`E`	Cache associativity.
`b`	$\log_2$ of the block size in bytes.
`hit_lat`	L1 hit latency in cycles.
`dram_read_lat`	DRAM read latency in cycles.
`dram_write_lat`	DRAM write latency in cycles.
`cpu_freq_ghz`	CPU frequency used when converting core cycles to nanoseconds.
`cache_freq_ghz`	Cache frequency used when converting cache-local cycles to nanoseconds.
`dram_freq_ghz`	DRAM frequency used when converting DRAM-local cycles to nanoseconds.

The framework core is untimed and in-order. Normal termination occurs either when an executed instruction is `ecall` or when execution falls through exactly to `text_end`. Misaligned instruction fetches, fetches outside `.text`, unsupported execution paths, and misaligned data accesses are treated as execution errors. The core also includes an internal step limit to catch infinite loops.

For every executed instruction, the framework appends one instruction-fetch request to the trace. Only `lw` and `sw` may append additional data requests. Trace entries are therefore byte-addressed requests of the form `I 0xADDR 4`, `L 0xADDR SIZE`, and `S 0xADDR SIZE`, stored in dynamic order. The framework owns instruction-fetch trace generation; your LSU owns data-request generation.

4 Memory System and Output Contract

The simulator models one shared cache and one blocking DRAM. The cache geometry is determined by 2^s sets, associativity E, and block size 2^b bytes. The store policy is fixed: the cache is write-back and write-allocate. A store hit marks a line dirty. A store miss first fetches the block into the cache and only then completes the store. Dirty data is written back only when a dirty line is evicted.

DRAM always uses block addresses rather than raw cache tags. In other words, the incoming request must be translated from its byte address into the correct block identity, and an evicted line must be translated back from cached metadata into the corresponding memory block before writeback. Thinking through that translation carefully is part of the homework. The cache uses deterministic LRU. On every hit or fill, the touched line becomes the most recent. An invalid line must always be used before evicting a valid line, and when a set is full the victim is the least-recently used valid line.

4.1 Statistics and Output

The simulator reports separate core, cache, and DRAM contributions. The number of core cycles is defined to equal the executed instruction count. `cache_cycles` is the total cache-local cycle cost accumulated during replay, and `dram_cycles` is the total DRAM-local cycle cost accumulated during replay. These local-cycle totals are converted into time with their corresponding configured frequencies:

$$\texttt{core\_time\_ns} = \frac{\texttt{core\_cycles}}{\texttt{cpu\_freq\_ghz}}, \quad \texttt{memory\_time\_ns} = \frac{\texttt{cache\_cycles}}{\texttt{cache\_freq\_ghz}} + \frac{\texttt{dram\_cycles}}{\texttt{dram\_freq\_ghz}},$$

and $\texttt{total\_time\_ns} = \texttt{core\_time\_ns} + \texttt{memory\_time\_ns}$. Every request, whether hit or miss, pays the cache hit latency. Misses then add incoming-read latency and dirty-writeback latency when applicable.

The meaning of the reported statistics is fixed and should not be reinterpreted. Table 4 summarizes the contract.

The final output printed to standard output must contain exactly these fields, one per line, in this order:

```
instructions: I
core_cycles: CC
cache_cycles: CAC
dram_cycles: DC
memory_requests: R
hits: H
misses: M
evictions: E
writebacks: W
core_time_ns: CT
memory_time_ns: MT
total_time_ns: TT
```

Table 4: Statistics contract

Field	Meaning
`instructions`	Number of executed instructions in the untimed framework core.
`memory_requests`	Number of logical trace requests, namely all instruction fetches plus all loads and stores.
`hits`, `misses`	Counted per logical trace request. A logical request is a hit only if every touched block hits.
`evictions`, `writebacks`	Counted per actual cache-line event, not per logical request.
`cache_cycles`	Total cache-local cycles consumed during replay.
`dram_cycles`	Total DRAM-local cycles consumed during replay.
`core_time_ns`, `memory_time_ns`, `total_time_ns`	Final time totals derived from the three local-cycle counters and the three configured frequencies.

5 Worked Example

Consider the following image:

```
entry 0x00001000
text 0x00001000 5
0x100000b7
0x0000a103
0x0020a223
0x00000193
0x00000073
data 0x10000000 2
0x11223344
0x00000000
```

These words correspond to `lui x1, 0x10000`, `lw x2, 0(x1)`, `sw x2, 4(x1)`, `addi x3, x0, 0`, and `ecall`. Assume the configuration $s = 0$, $E = 2$, $b = 2$, `hit_lat = 1`, `dram_read_lat = 10`, `dram_write_lat = 20`. The configured frequencies are CPU 2.0 GHz, cache 1.0 GHz, and DRAM 0.5 GHz. This particular choice gives a small one-set, two-line cache so that shared-cache interference and LRU replacement are both visible in a short example. The example is illustrative rather than exhaustive; the same public contract also covers direct-mapped and larger multi-set organizations.

During execution, the framework core contributes five instruction-fetch requests, and your LSU contributes two data requests. The unified trace is therefore:

```
I 0x00001000 4
```

```
I 0x00001004 4
L 0x10000000 4
I 0x00001008 4
S 0x10000004 4
I 0x0000100c 4
I 0x00001010 4
```

Replay begins with `instructions = 5`, `core_cycles = 5`, and `memory_requests = 7`. Because every request maps to the same set, the cache repeatedly encounters conflict misses, eventually evicting a dirty store line. The final statistics are shown in Table 5.

Table 5: Worked-example final report

Field	Value
instructions	5
core_cycles	5
cache_cycles	7
dram_cycles	90
memory_requests	7
hits	0
misses	7
evictions	5
writebacks	1
core_time_ns	2.500000
memory_time_ns	187.000000
total_time_ns	189.500000

You can run this deterministic example locally with:

```
make test
```

The public `test` target compares your simulator output against the checked-in example report. It is a sanity check, not the full grading suite. It is still useful because it checks the required output format and the deterministic worked-example totals in one place.

6 Final Remarks

Most of this guide focuses on the regular case where one logical request maps cleanly to one cache block. If the block size is small enough, however, one aligned access may still span more than one cache block. For this release, treat that split-line behavior as an advanced edge case rather than the main baseline. The starter provides basic block-address helpers in `cache.c`, but no dedicated split-request helper. Stronger solutions can therefore extend `mem_ctrl.c` to service all touched block-local accesses in order while still reporting one logical `memory_request`.

Before submitting, make sure that your simulator preserves the fixed header interfaces, prints exactly the required report format on standard output, and does not emit extra debugging

text on standard output. Keep your solution self-contained under the submitted tree rather than depending on files outside the submission root. The required simulator output includes timing-related fields, so `mem_ctrl.c` must compute the correct cache-side and DRAM-side contributions rather than reaching the final totals only by accident. Use framework-owned modules only through their declared interfaces rather than bypassing `trace`, `cache`, `dram`, `replay`, or `core` with private parallel mechanisms. The implementation effort for this release should remain concentrated in `src/hw/lsu.c`, `src/hw/cache.c`, and `src/hw/mem_ctrl.c`.