154B Discussion 6

February 15th, 2023

Goals

- Assignment 4
  - Multi-cycle memory components
  - Caching
- Caches
- Week 6 Quiz.

Logistics

- Assignment 3 due on Friday.
- Assignment 4 is going to be released sometime this week.

Assignment 4

- More complete system compared to previous assignments
  - 5-stage pipelined cpu, the same one as in assignment 3.
  - Memory request no long takes one cycle to complete.
  - One level of memory cache, with at least 3-cycle latency.
  - RAM device now has 30-cycle latency.

Split L1 cache: L1 Instruction Cache (L1I) & L1 Data Cache (L1D)

L1I and L1D are similar:
each of them has 32 entries, 4-way associative, block size = 8 bytes, LRU replacement policy.

Assignment 4: Motivation

- Multi-cycle memory component: a memory component that takes more than one CPU cycle to process a memory request.
- Expecting a memory request to be complete within 1 cycle is unrealistic.
- Real systems have caches.

Assignment 4: Motivation

Mem Hierarchy	AMD EPYC 7742 DDR4-3200 (ns @ 3.4GHz)	AMD EPYC 7601 DDR4-2400 (ns @ 3.2GHz)	Intel Xeon 8280 DDR-2666 (ns @ 2.7GHz)
L1 Cache	32KB 4 cycles *1.18ns*	32KB 4 cycles *1.25ns*	32KB 4 cycles *1.48ns*
L2 Cache	512KB 13 cycles *3.86ns*	512KB 12 cycles *3.76ns*	1024KB 14 cycles *5.18ns*
L3 Cache	16MB / CCX (4C) 256MB Total ~34 cycles (avg) *~10.27 ns*	16MB / CCX (4C) 64MB Total	38.5MB / (28C) Shared ~46 cycles (avg) *~17.5ns*
DRAM **128MB Full Random**	~122ns (NPS1) ~113ns (NPS4)	~116ns	~89ns
DRAM	~134ns (NPS1)		~109ns

https://www.anandtech.com/show/14694/amd-rome-epyc-2nd-gen/7

Assignment 4

Part 1: Re-implement the hazard detection unit.

Part 2: Collecting data.

Part 3: Performance Analysis: Explain why there's a speedup or a slowdown.

Start early! Correctness of part 2 & 3 are dependent on how reasonable part 1 is implemented.

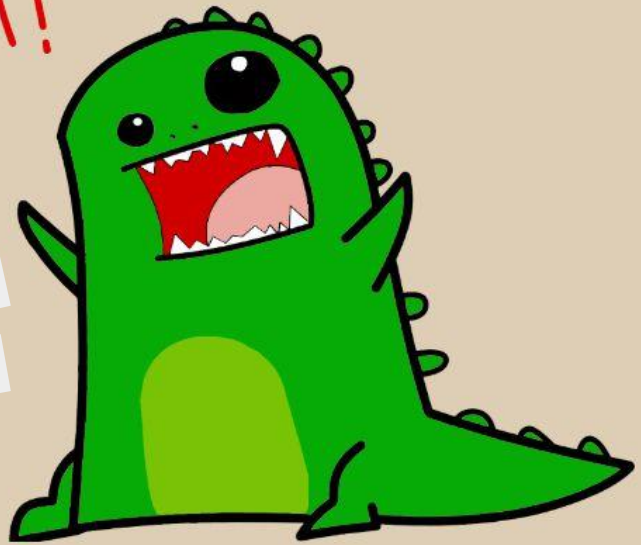
Assignment 4: Re-implement the hazard detection unit

RAWЯ!

it means

new data hazard

in DINO CPU




RAWR!
it
means
new data hazard
in DINO CPU

Assignment 4: Re-implement the hazard detection unit

- Data hazards,
  - We’re not re-ordering the instructions so data hazards are still the same as the ones in assignment 3.
  - Still have RAW hazards (resolved by forwarding) and load-to-use hazards (resolved by stalling).
  - Don’t need to modify forwarding unit.

Assignment 4: Re-implement the hazard detection unit

- Control hazards,
  - Still have to take care of mispredictions, i.e. jumps and taken branches.

Assignment 4: Re-implement the hazard detection unit

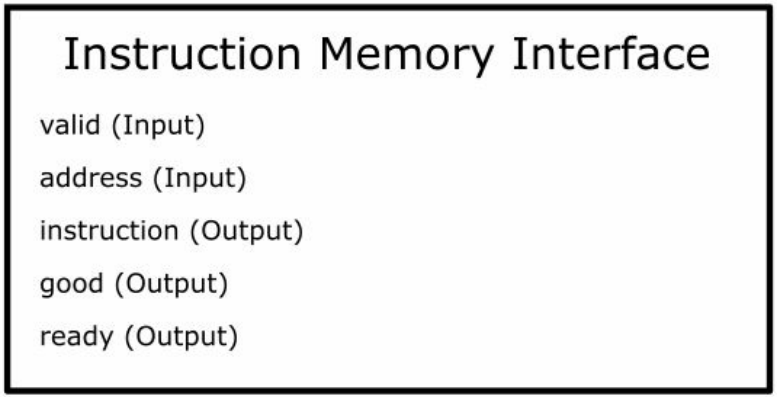
- There’s no new type of hazard. However, now that timing involves, we have to take care control hazards quite differently.

Assignment 4: Re-implement the hazard detection unit

- There’s no new type hazard. However, now that timing involves, taking care of control hazards are more involved.

Assignment 4: The new memory interface

- Both instruction memory and data memory interfaces now have additional inputs and outputs,
  - valid (CPU inputs to memory): 1 if CPU sends the request, 0 otherwise.
  - good (Output from memory): the response is valid.
  - ready (Output from memory): the memory is ready to receive memory request.



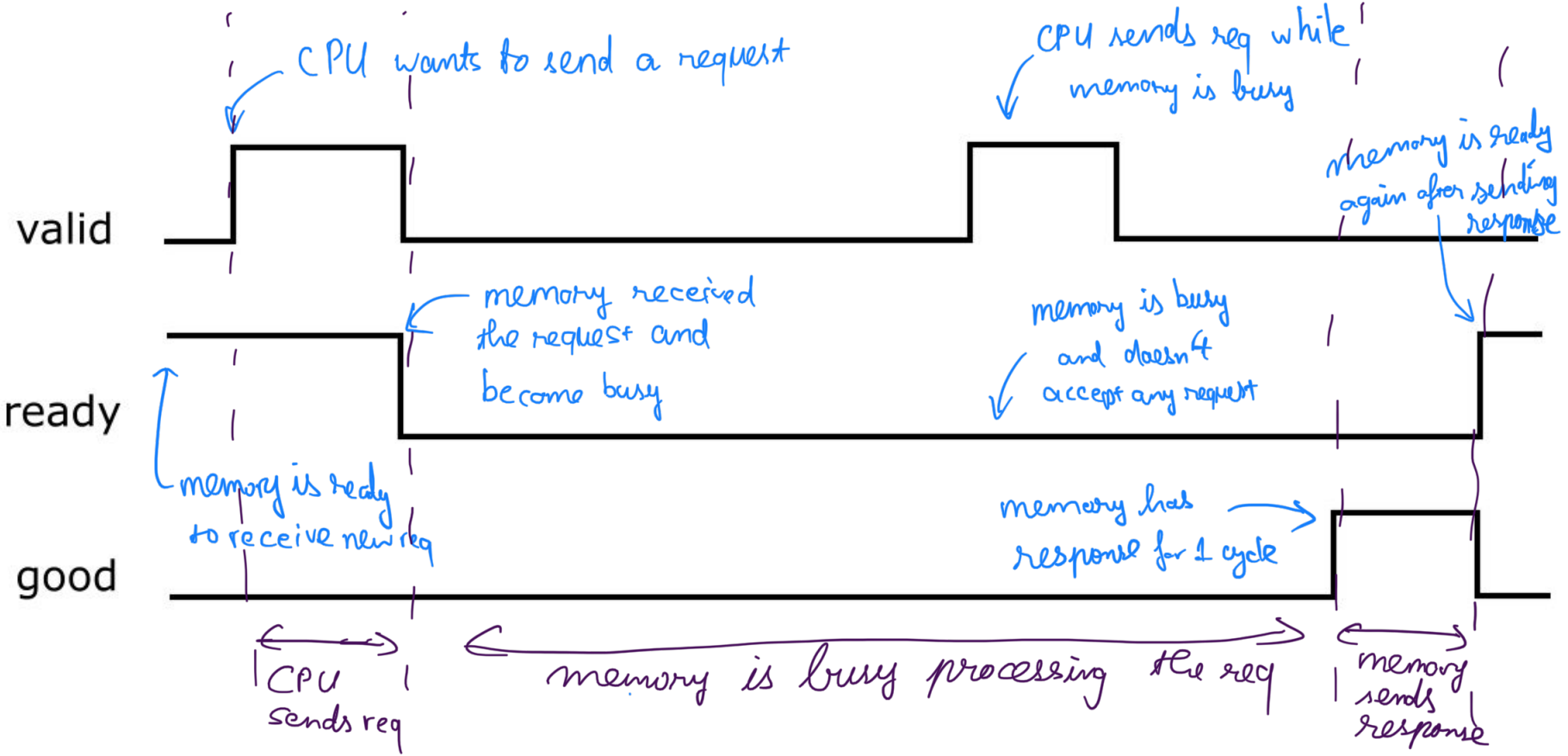
Assignment 4: The new memory interface

- The contract between the memory interface and the CPU,
  - The CPU only sends a request when valid is true and Memory is ready.
  - The memory only processes one request at a time.
  - If the memory receives a request in the current cycle, it will start processing the request in the next cycle.
  - If the memory is not ready, then even if the CPU sets valid to 1, the memory does not receive the request.
  - The memory only guarantees the response to be correct when and only when the `good` signal is 1. It means, if the `good` signal is 0, the response might contain garbage.
  - The `good` signal is only set to one for 1 cycle. It means, the response is only guaranteed to be correct in that cycle.

Assignment 4: The new memory interface

- Note: there’s no guarantee in terms of memory request latency, and there’s no upper bound for the latency of processing one request.
- Should rely on the good/ready signals rather than stalling the pipeline for a fixed number of cycles.

Assignment 4: The new memory interface



Assignment 4: Where are the valid instructions

- Recall: we use IF_ID, ID_EX, EX_MEM, and MEM_WB stage registers to store the instructions.
- The output of IF_ID contains the instruction in ID stage.
- The output of ID_EX contains the instruction in EX stage.
- The output of EX_MEM contains the instruction in MEM stage.
- The output of MEM_WB contains the instruction in WB stage.
- Where is the instruction in the IF stage?
  - In assignment 3, the instruction in the IF stage is in the instruction memory!
  - The instruction in the IF stage does not necessarily enter the pipeline!
- Only when the instruction reaches ID stage, it enters the pipeline.

Assignment 4: Where are the valid instructions

- Now that the output of the instruction memory is not always a correct instruction, we don't want to advance the instruction in the IF stage to the ID stage if the response from instruction memory is not "good".
- How does that look like? It's quite hard to keep the pipeline busy!

```
                          |  IF  |  ID  |  EX  | MEM  |  WB  |
cycle 15, imem.good = 0 | err  | add  | ---  | ---  | ---  |
cycle 16, imem.good = 0 | err  | ???  | add  | ---  | ---  |
cycle 17, imem.good = 0 | err  | ???  | ???  | add  | ---  |
cycle 18, imem.good = 1 |  ld  | ???  | ???  | add  | ---  |
cycle 18, imem.good = 0 | err  |  ld  | ???  | ???  | add  |
```

Assignment 4: Re-implement the hazard detection unit

- HazardUnit has the power of stalling/flushing the pipeline. We can use it to decide when to advance an instruction to the pipeline.
- Other issues involving timing will be discussed in later discussions.

Assignment 4: Benchmarks

- There are four benchmarks
  - stream-64-stride-1.riscv
  - stream-64-stride-4.riscv
  - stream-64-stride-1-noverify.riscv
  - stream-64-stride-4-noverify.riscv
- The latter two (ones without verify functions) are used for performance evaluation.
- The first two are used for testing to make sure that the data when running the latter two are reasonable.

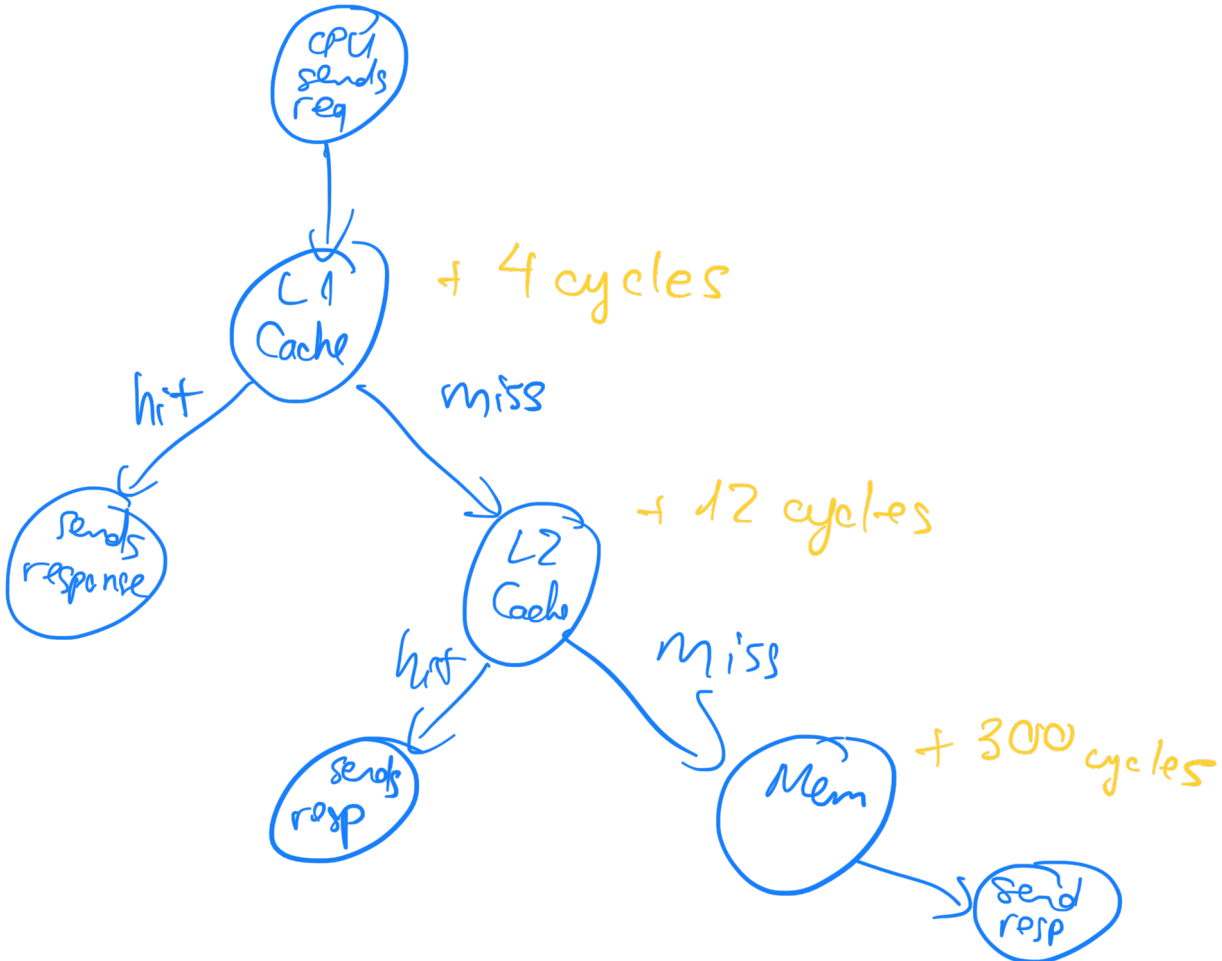
The benchmark looks like

```
uint16_t *src, *dst;
for (size_t iter = 0; iter < 2; iter++)
    for (size_t idx = 0; idx < 64; idx++)
        dst[idx] = src[idx]
```

```
uint16_t *src, *dst;
for (size_t iter = 0; iter < 2; iter++)
    for (size_t idx = 0; idx < 256; idx += 4)
        dst[idx] = src[idx]
```

Caches

- Diagram



Caches: AMAT (average memory access time)

AMAT = L1 access time (4 cycles) + L1 miss rate × L1 miss time

where L1 miss time = L2 access time (12 cycles) + L2 miss rate × L2 miss time

where L2 miss time = main memory access time (300 cycles)

Generalization with N-level cache:

AMAT = L1 access time + L1 miss rate × L1 miss rate

where
- L_j miss time = $L_{(j+1)}$ access time + $L_{(j+1)}$ miss rate × $L_{(j+1)}$ miss time, if L_j is not the last level cache (LLC)
- L_j miss time = main memory access time if L_j is LLC

Caches: Types

- Direct mapped (one entry per possible index).
- N-way associative (N entries per possible index).
- Fully associative (no index).

entries = # rows × # ways

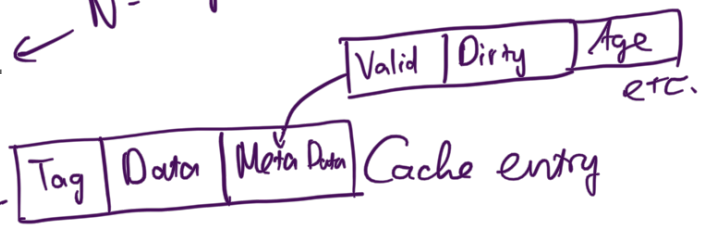
→ # rows = # entries / # ways

(for fully associative cache, # rows = 1, for others, # rows = # different indices)

1-way cache

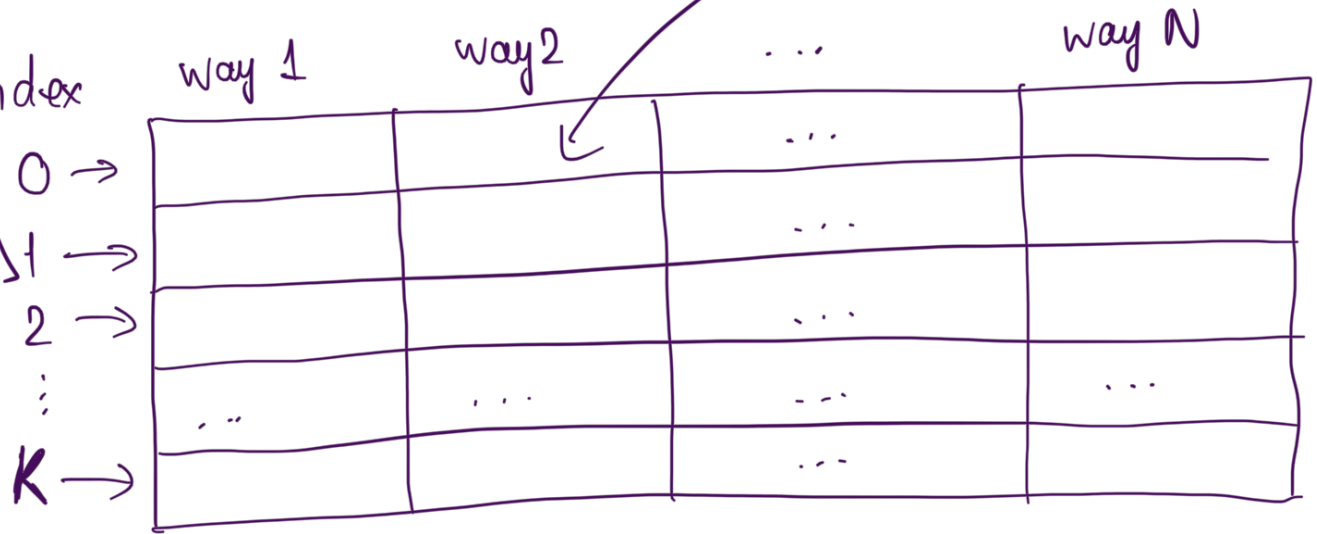
N-way associative cache

1 row



Cache organization:

- each cache entry must have enough information to uniquely identify the address of the data.
  - ↳ address = [Tag | Index | ...]



Caches: Tag bits, Index bits, Offset bits

Address = [Tag bits | Index bits | Offset bits]

offset bits = $\log_2$ (cache block size in bytes)

index bits = $\log_2$ (# rows)

tag bits = # address bits − # offset bits − # index bits

↳ a cache entry must have tag to recover the tag bits

- the index bits can be recover from the index of the entry in the cache table (i.e., which row)
  ⇒ we know whether the cache block contains the requested address

Caches: Cache Block Size

- An entry of a cache holds <cache_block_size> amount of data.
  - ↳ in real systems: ~ 64 bytes / block
  - ↳ in DINO CPU : ~ 8 bytes / block
- Cache block size determines the number of offset bits = $\log_2$ (cache block size in bytes)
- E.g., req address = 0x104, block size = 64 bytes → 6 bits for offset

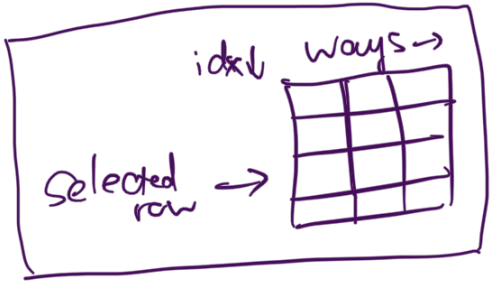
  0001 0000 0004
  offset

  ↳ the cache block will contain the data of address in the range: 0001 0000 0000 → 0001 0 111 1111

  (0x100 → 0x17F)

Quiz 6

Where to place a new cache block?



new cache block (Data + address)

↓

select the row according to the index bits

↓

Compare the tags of entries in that row

↓ if there's no identical tag

is there an entry in that row that is not valid?

yes ↙

write to the invalid entry

no ↘

evict an entry from that row according to <replacement-policy>

↳ write to the evicted entry

if there's an identical tag ↙

write to that block

Quiz 6

LRU replacement policy: the cache will evict the entry with the oldest age.
↑ least recently used

- When the age of an entry is updated?
  → upon a memory access (read from / write to that block)

Quiz 6

. 1920 × 1080 pixels,

a. 1 bit / pixel → $\frac{1920 \times 1080}{8}$ bytes

b. 3 bytes / pixel → 1920 × 1080 × 3 bytes

c. 90 Hz → transferring data 90 times / second

→ bandwidth = 1920 × 1080 × 3 × 90 bytes / second

→ MiB / second

(round to the nearest integer)

d. Watts = Joules / second ; 2×10^{-9} J/bit × bandwidth (in bits / second)