154B Discussion 5

February 11th, 2022

Goals

- Overview of Assignment 4.
- Some memory system stuff (AMAT, memory array calculation)

Assignment 4: Overview

- Implement forwarding logic and hazard detection logic for dual-issue pipelined CPU.
- Run full applications with/without loops unrolled on implemented CPU designs.
- Reason about the performance of different CPU designs on different workloads.

Assignment 4: Dual-issue Pipelined CPU

- Issuing: occurs in the Fetch stage, decides how many instructions will enter the pipeline in that cycle.
  - Different from fetching, which means loading instructions from memory.
  - Can only issue instructions fetched from memory.
- Multiple-issue: issuing multiple instructions in a cycle.
  - Dual-issue: issuing *upto* 2 instructions per cycle.

Assignment 4: Why Dual-issue?

- Further exploit the idea of instruction-level parallelism.
  - Maximizing the number of instructions could be executed at the same time.
- For DINOCPU, in terms of number of cycles need to complete a program, dual-issue design always needs fewer cycles than the original pipeline.

Dual-issue in Real World

- Used in ARM's low power core Cortex A55 (2017).

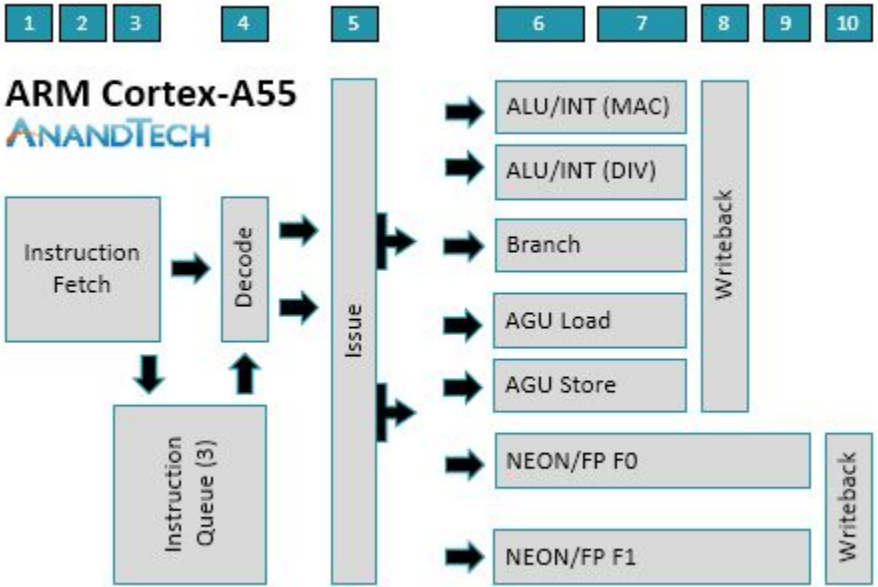
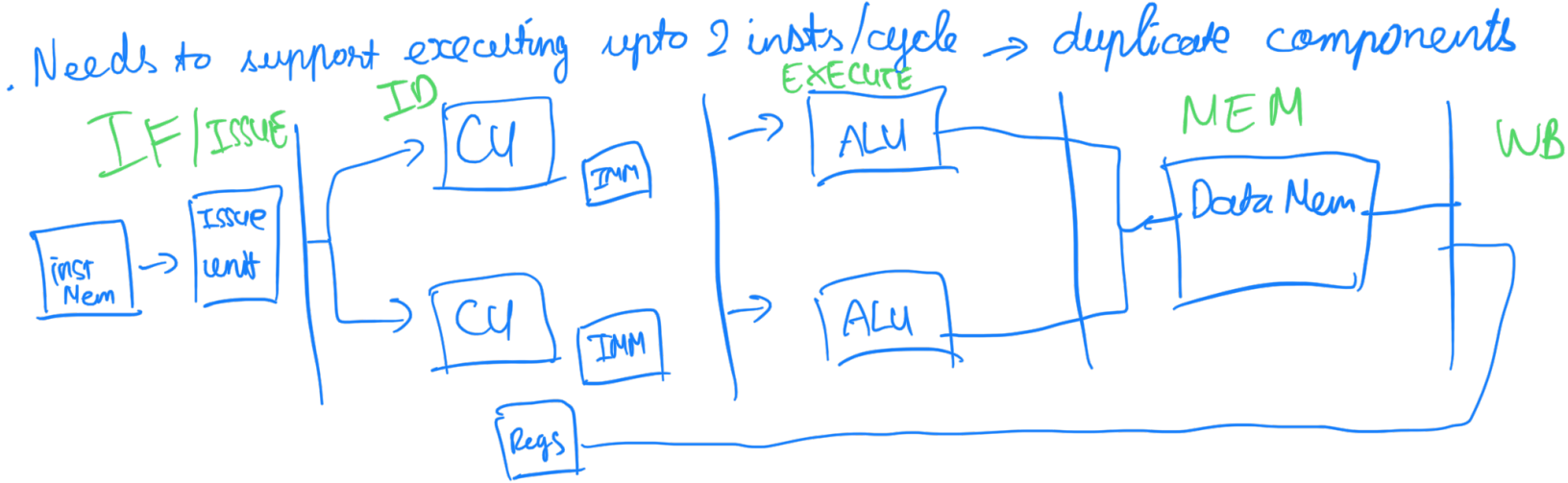


Image: https://www.anandtech.com/show/11441/dynamiq-and-arms-new-cpus-cortex-a75-a55/4

Assignment 4: How to implement dual-issue pipeline?

- Duplicate almost every component and all stage registers.
- Modifying forwarding unit and hazard detection unit.



Assignment 4: How fetched instructions are issued?

- There are two pipelines in dual-issue design: pipeA and pipeB.
  - If the issue unit decides to issue 2 instructions, the instruction at PC will be issued in pipeA, and the instruction at PC+4 will be issued in pipeB.
  - If there is only one instruction being issued, that instruction will enter pipeA, and the instruction in pipeB will be marked as invalid.
- Issue constraints:
  - The issue unit will only issue 2 instructions if,
    - The instruction at PC is not a load/store/jump/branch.
    - The instruction at PC will not write to a register that will be read by instruction at PC+4.
- The constraints lead to this property,
  - If the instruction at a stage in pipeA can update PC, the instruction at that stage in pipeB is invalid.
    - → that instruction should not be executed

BRANCH / JUMP

Assignment 4: Loop Unrolling

- Compiler optimization option, trade-off between the binary size and potential instruction-level parallelism.
- Main idea: doing more work per iteration of a loop.

```
// A loop
for (int i = 0; i < 16; i++)
{
  arr[i] = c[i] + 1;
}
```

Before being unrolled

```
// Unrolled loop
for (int i = 0; i < 16; i+=4)
{
  arr[i] = c[i] + 1;
  arr[i+1] = c[i+1] + 1;
  arr[i+2] = c[i+2] + 1;
  arr[i+3] = c[i+3] + 1;
}
```

fewer iterations

more work per iteration

Assignment 4: Loop Unrolling

dinocpu branch execution: speculatively execute the branch as if the branch is not taken.

- Loops, in machine code.

```
// A loop
label1:
addi a1, a1, 1
// do 1 thing
blt a1, 16, label1
label2:
…
```

a1 → current index

← this is the loop branch executed 16 times
→ not taken: 1 time
→ taken: 15 times

Wasted cycles: 15 x 3 = 45 cycles

```
// Unrolled loop
label1:
addi a1, a1, 4
// do 1 thing
// do 1 thing
// do 1 thing
// do 1 thing
blt a1, 16, label1
label2:
…
```

← loop branch executed 4 times
→ not taken: 1 time
→ taken: 3 times

→ # wasted cycles: 3 x 3 = 9 cycles

Assignment 4: Loop Unrolling

- In the assignment, the binaries with loops unrolled only have the loops of known size unrolled.
- What happens when the size is unknown?

```
// A loop

while (cond)
{
  // do something
}
```

```
// Unrolled unknown size loop, machine code

label1:
// do 1 thing
blt a1, a2, label1
// do 1 thing
blt a1, a2, label1
// do 1 thing
blt a1, a2, label1
// do 1 thing
blt a1, a2, label1

label2:
...
```

have to check stopping condition multiple times per iteration

AMAT: Average Memory Access Time

- 1-level cache system

$$AMAT = L1\_latency + (1 - L1\_cache\_hit\_rate) * memory\_latency$$

- 2-level cache system

$$AMAT = L1\_latency + (1 - L1\_hit\_rate) \times [L2\_latency + (1 - L2\_hit\_rate) \times memory\_latency]$$

- 3-level cache system

$$AMAT = L1\_latency + (1 - L1\_hit\_rate) \times (L2\_latency + (1 - L2\_hit\_rate) \times (L3\_latency + (1 - L3\_hit\_rate) \times memory\_latency))$$

- Generalized: n-level cache:

$$AMAT = L1\_latency + (1 - L1\_hit\_rate) \times AMAT_{L2}$$

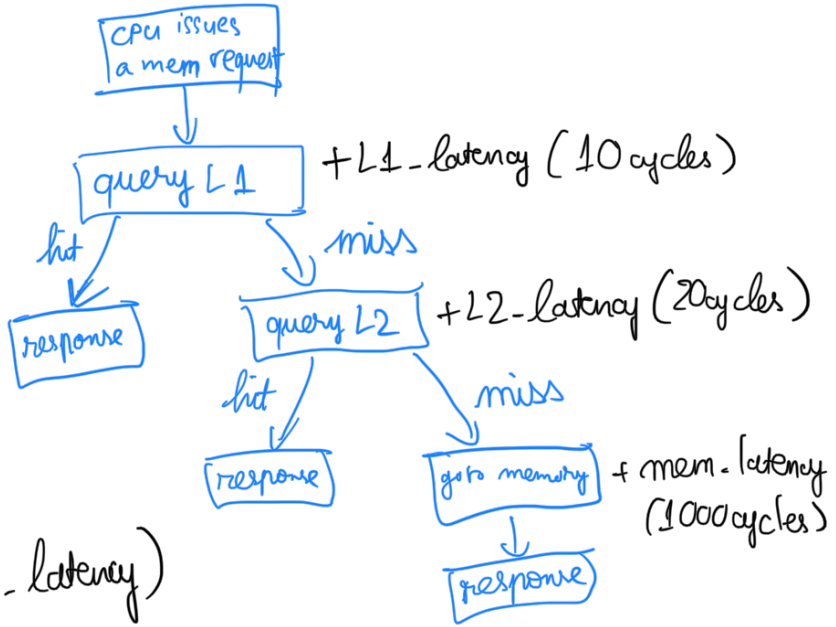
$$AMAT_{L2} = L2\_latency + (1 - L2\_hit\_rate) \times AMAT_{L3}$$

$$\vdots$$

$$AMAT_{Ln} = Ln\_latency + (1 - Ln\_hit\_rate) \times memory\_latency$$

AMAT: Example

- 2-Level Cache System
  - L1 hit rate: 30%; L1 latency: 10 cycles
  - L2 hit rate: 60%; L2 latency: 20 cycles
  - Memory latency: 1000 cycles



From graph:

AMAT = L1-latency + (1 - L1-hit-rate) × (L2-latency + (1 - L2-hit-rate) × mem-latency)

= 10 + (1 - 0.3) × (20 + (1 - 0.6) × 1000)

= 304 cycles

general idea:

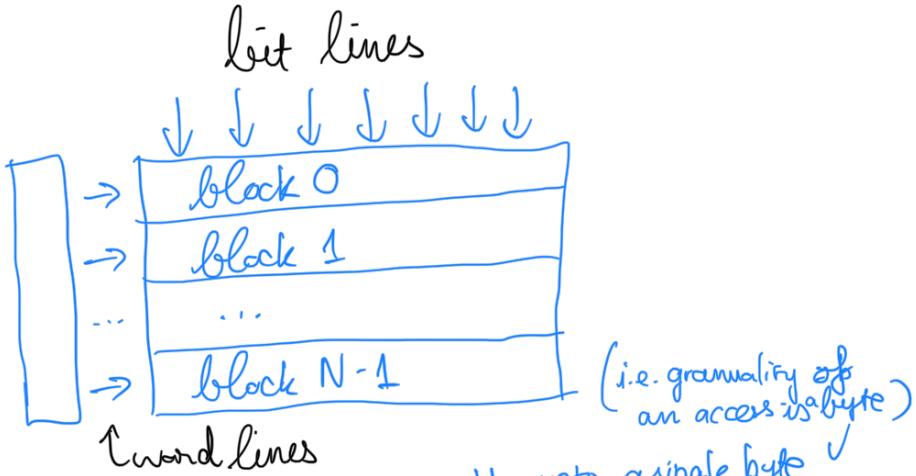
→ accessing cache or memory costs time regardless whether it results in a hit or a miss.

→ if an access is a miss, needs to query the next level

Memory Array

- Row: Word lines
- Column: Bit lines
- Address: (row bits) (column bits)

Mem Array:
→ N different rows
→ each row has 1 block of M bits
→ bit lines: selecting columns
→ word lines: selecting 1 row
→ to get data: select a row first, then select columns



→ # of bits for selecting a row:
log2(how many different rows) = log2(N)

→ # of bits for selecting columns assuming byte addressable:
log2(how many possibilities for selecting columns)
= log2(M/8) = log2(M) − log2(8)
= log2(M) − 3

• byte addressable: memory is accessible upto a single byte

Memory Array: Example

- Memory Array
  - Byte-addressable
  - 4096-bit word per block
  - Capacity: 262144 bits

LED question

. refresh rate of 90 Hz → the screen needed to be updated 90 times per second (or every 1/90 sec)

. Watt = Joule / second

Bandwidth: $1920 \times 1080 \times 24 \div 8 \div 1024^2 \times 90$

$1920 \times 1080 \times 24$: # of bits of framebuffer

$\div 8$: bits → bytes

$\div 1024^2$: bytes → MiB

$\times 90$: needs to update the framebuffer 90 times / second

$1920 \times 1080 \times 24 \div 8 \div 1024^2$: MiB needed per update

Power (Joule / second) $= 1920 \times 1080 \times 24 \times 90 \times 2 \times 10^{-9}$

$1920 \times 1080 \times 24$: # of bits per update

90: # updates / second

2×10^{-9}: 2 nJ per bit, nJ → J