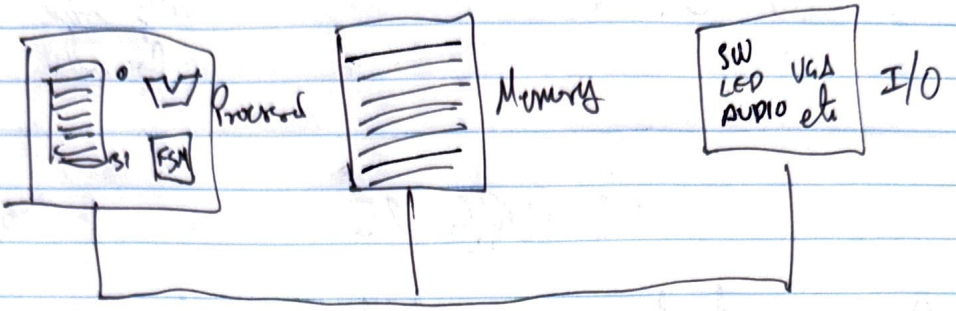
LECTURE 24

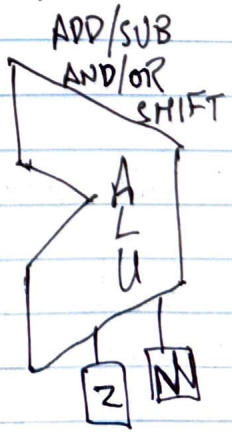
06/03

Computer Architecture So Far....



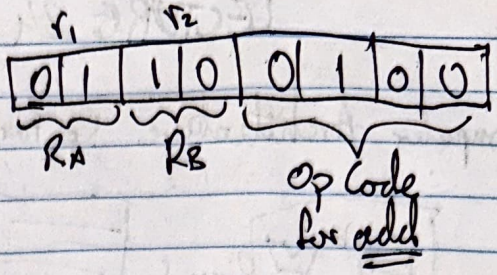
ECE243 Simple Processor

- Four registers $r_0 \rightarrow r_3$, word size 8 bits
- 256 bytes of memory, has a program counter
- Only 10 instructions → each takes two arguments
- NEW → Z-bit = 1 if result of most recent computation is zero
  - ↳ N-bit = 1 if result of most recent computation is negative
  - ↳ Used in conditional branch instructions
- Consider add instruction → add r_1, r_2 $r_1 \leftarrow r_1 + r_2$



① Fetch encoded instruction from mem
② Figure out what instruction is
③ Compute $R_S \leftarrow R_A + R_B$
④ Set N & Z
⑤ Increment pc

• Instruction encoding →



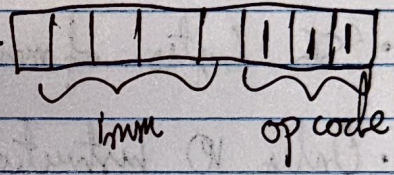
0x64 ↙

LECTURE 25 10/03

• OR-immediate → immediate is 5 bits → no room to specify a register

↳ So we always store into R1 and zero-extend the immediate

We reserve two 4-bit opcodes (0111) & (1111) to ORI



Conditional instructions

• In Simpler we use result of prev operation to compare to zero

↳ bnz imm4 → execution → if (z == imm4) PC += SE(imm4)
else PC += 1

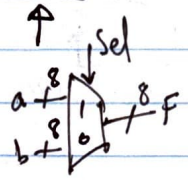
• Relative branching gives us movable code

LECTURE 26 12/03

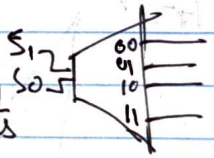
* Let's write a small assembly program. (we are assembler)

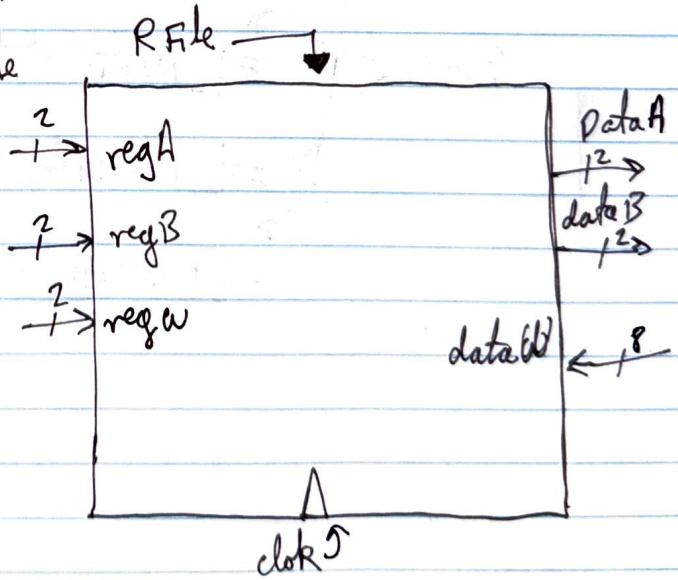
↳ subtracts 2 numbers and branches back to the beginning

Address		Instruction		Encoding	Hex
0	loop:	load R_2,(R0)	#$R_2 \leftarrow$ mem(R_0)	10000000	0x80
1		load R_3,(R1)	#$R_3 \leftarrow$ mem(R_1)	11010000	0xA0
2		sub R_2,R_3	#$R_2 \leftarrow R_2 - R_3$	10110110	0xB6
3		bnz loop		11011001	0xD9

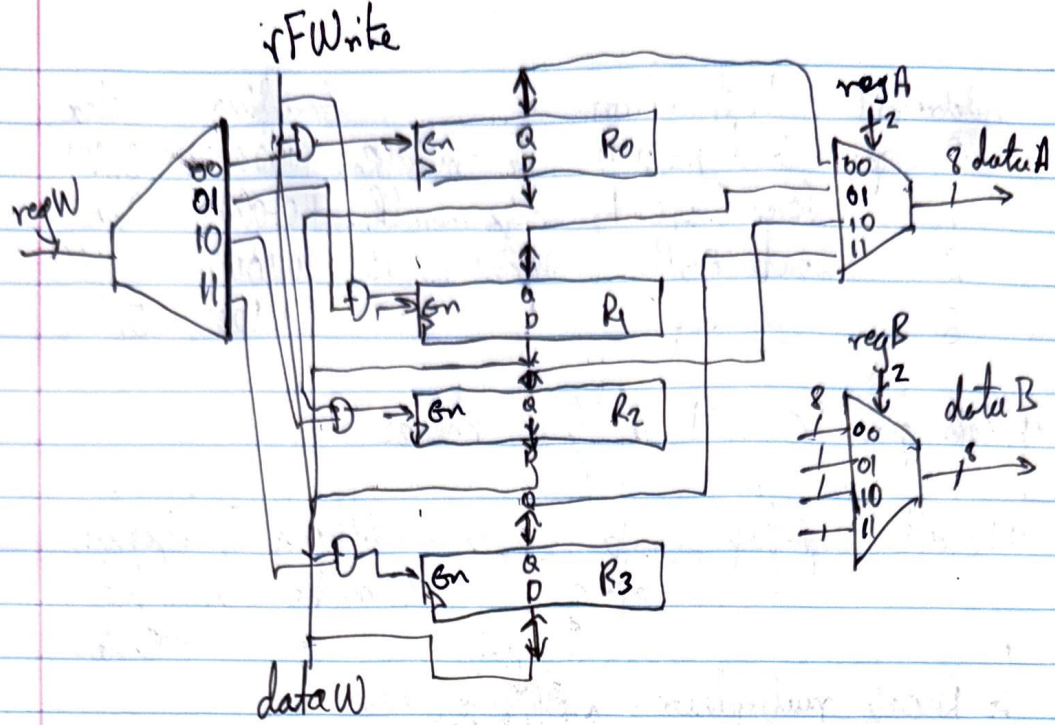
- Recall from ECE241 a few things
  - Recall flip-flop  → Attach 8 together to make an 8-bit register
  - Recall multiplexer 

LECTURE 27 13/03

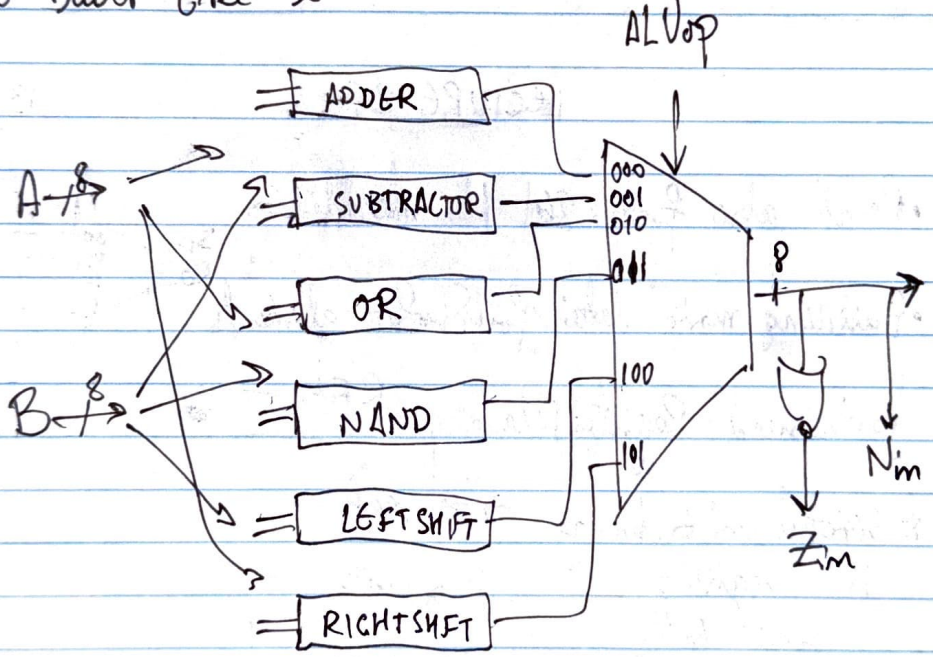
- Recall also from 241 the decoder → 
- Building more detail for SimProc elements
  - We need Register File



* Careful not to change the registers accidentally

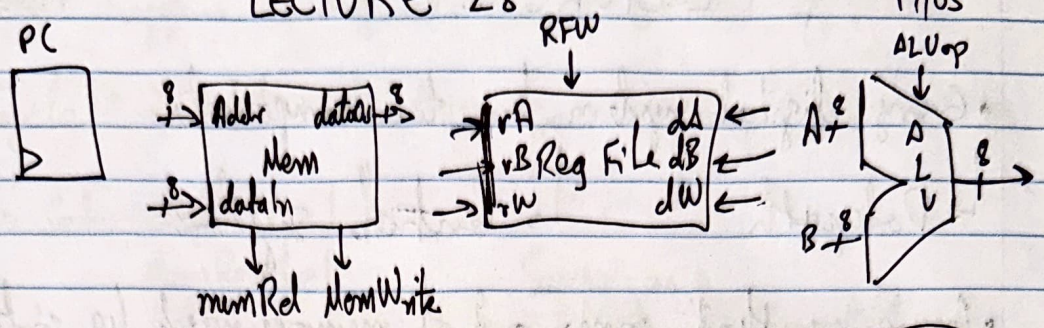


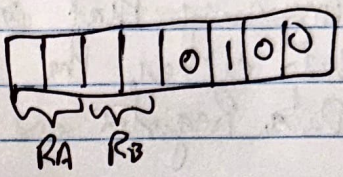
ALU Built Like So



LECTURE 28

17/03





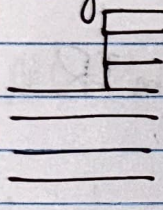
- Here's what happens in full execution
  - ① Fetch opcode from memory
    - Put this into the instruction Register
  - ② Decode the instruction → Stored & decoded in IR
  - ③ Read R_A & R_B from register
  - ④ Compute ALUout ← $R_A + R_B$
  - ⑤ Write ALUout back into R_A
  - ⑥ Set Zin & Nin based on the operation's result
  - ⑦ Compute PC = PC + 1

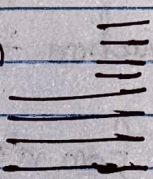
☆ Special case happening w/ ORI instruction

↳ Look at last 3-5 mins of lecture to review

LECTURE 29 — 19/03

- Every digital system has two components
  - ↳ Datapath
  - ↳ Control system
- Anything that comes out of memory must be sorted into either the Instruction Register or Memory Data Register
- Dereferencing data / storing into memory done by interfacing w/ Addrsel & the memsel mux.


Sign extension


Zero extension

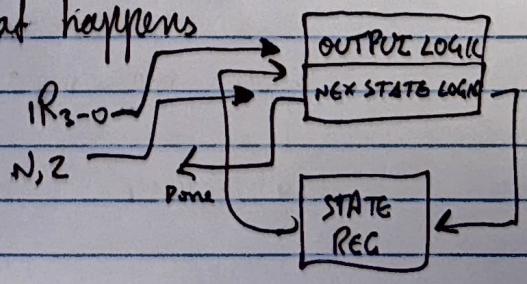
- How does IR know if opcode is 3 or 4 bits?
  - ↳ SimProc automatically responds to XOII as a shift regardless of what X is

---

LECTURE 30 — 20/03

Finite State Control of SimProc

- FSM controls which signals are activated & when (on which clock signal) that happens



• What happens during each cycle for ADD/SUB/NAND

① Cycle 1 → Fetch instruction from memory: IR ← Mem(pc)

↳ Set AddrSel = 1
MemRead = 1
IRLoad = 1

Saves us a cycle of compute!

↳ We can also get PC ← PC+1 in the same cycle

↳ Set ALU_A = 0 (PC → ALU)
ALU_B = 001 (ALU_B → 1)
ALU_op = 000 (add)
PCwrite = 1

Clock-to-Q time ensures we don't screw up here
↳ setup & hold time
↓
Shows up in many interviews

② Cycle 2 → Decode the signal in IR (needs 1 full cycle)

↳ We guess we'll probably need to read R_A / R_B

↳ Set R_Asel = 0 (selects R_A to regA)
ABLD = 1

③ Cycle 3 → Add / Sub / NAND specific

↳ Set ALU_A = 1 (select A to ALU)
Set ALU_B = 000 (select B to ALU)
set ALU_op = one of add/sub/nand
ALU_outLD = 1
flagWrite = 1

③ Cycle 4 → Get ALU_out to R_A

↳ Set RegIn = 0 (connect ALU_out → dataW)
RFWrite = 1
Done = 1 → This is for next-state logic

Now lets do "load R_a, (R_B)"

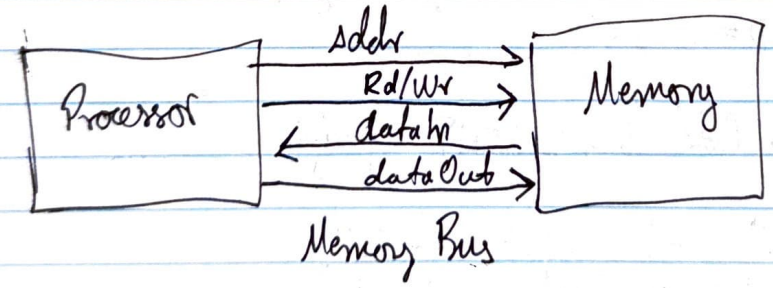
↳ Only need cycle 3 onwards

③ Cycle 3 → AddrSel = 0
MDRload = 1
MemRead = 1

④ Cycle 4 → RegIn = 1 (MDRload → dataW
RFWrite = 1
Done = 1

LECTURE 31 24/03

- Introduction to Cache Memory



Processor has two memory access types

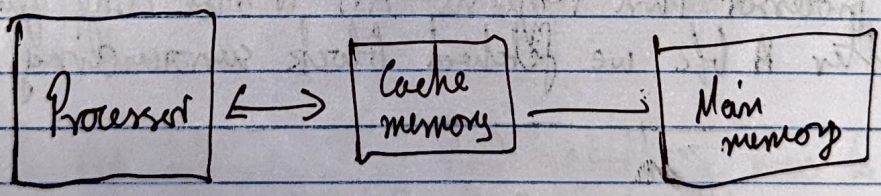
↳ Read (memory or instructions) → both look the same to memory

↳ Write (store)

- Some physics for context:
  - Larger memories are slower ↔ smaller memories faster
    - Longer wires have more resistance & capacitance, making signal propogation slower
    - Decoder time will get slower in larger memories
- Interesting point is that as chips get physically bigger they don't necessarily get slower
- 16 GB memory system takes ~300 cycles to respond to memory-access request

How to solve the problem of a fast processor but a big, slow memory?

- Put a small fast memory in between the processor and main memory & keep the much smaller active memory in between.
  - Called the cache memory



- Sets of instructions and certain data structures benefit from having close addresses
  - Locality of reference:
    - Some set of instructions called multiple times

- Two types of locality
  - ↳ Spatial locality → physically closer together
  - ↳ Temporal locality → Addresses come over time

Operation of a Memory Read Using a Cache

① Processor requests to read data from address A.

② Cache (somehow) looks inside to see if it has a copy of data at address A.

- ↳ If no, we call a cache MISS. Cache requests data from A (and some chunk of block).
  - ↳ Memory responds slowly, cache copies to itself and sends to the processor
- ↳ If yes, then we call it a cache HIT. Quickly send that data to the processor in just a few cycles

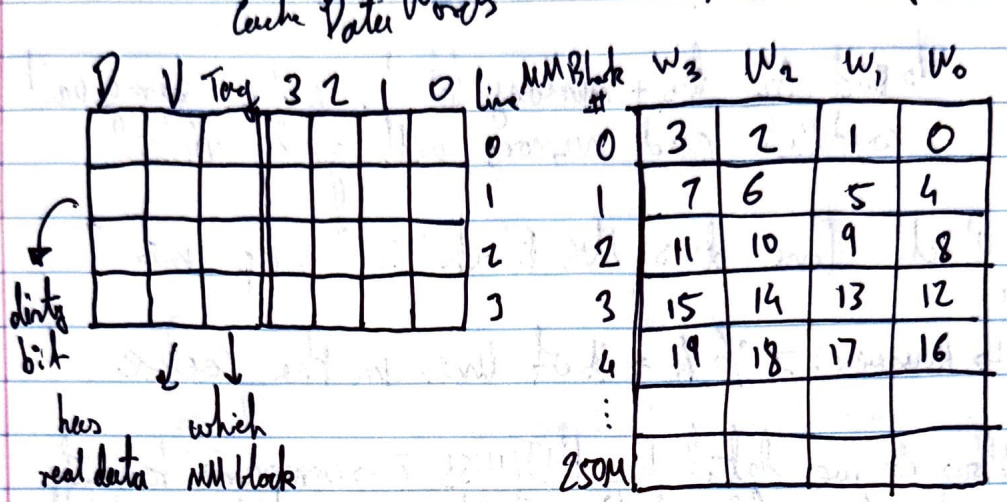
③ If processor then requests A+1 it will come quickly after A b/c we fetched block surrounding A.

LECTURE 32 26/03

Logistics of the Cache Memory

① Because cache asks for >1 mem location when MISS, we consider main memory as organized into blocks (say 4 words for eg).

- We assume every word has an address



The Direct-Mapped Cache

- Cache has rows & columns like MM, each row called cache line

↳ line size equals main memory block size b/c we take whole blocks into cache lines

- Tag is group of bits that tell you which MM block is in the cache line

- Valid bit tells us if there's real data in the cache or not

- Dirty bit tells us, if we write something new to cache, needs to send information back to MM.

- Now consider when an address is requested by the processor → where can the data go?

↳ 1 line → "Direct-Mapped" Cache

↳ good bc we only have to look at 1 tag/line in cache for HIT/MISS

↳ Bad bc two + memory blocks using same will conflict and memory will get evicted

Which line does MM Block "J" map into?

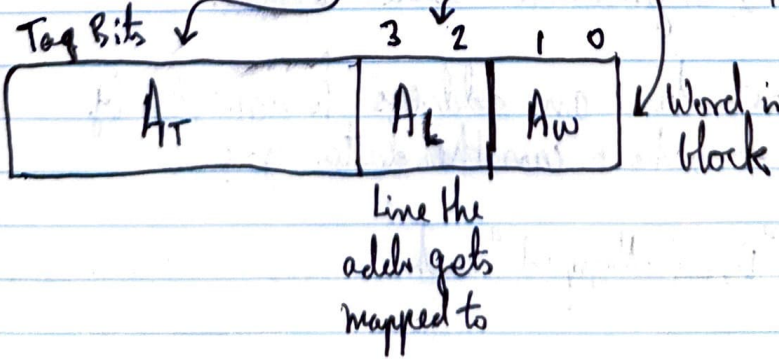
↳ Answer: J % 4 → # of lines in the cache

How do we detect the HIT/MISS? → compare tag of cache line (if valid) w/ the tag of requested memory block.

Consider the possible memory access request addresses

Block #	Address Bits (31 30 ... 5 4 3 2 1 0)	
0	0 0	0 0 0 0 X X
1	0 0	0 0 0 1 X X
2	0 0	0 0 1 0 X X
3	0 0	0 0 1 1 X X
4	0 0	0 1 0 0 X X
5	0 0	0 1 0 1 X X
6	0 0	0 1 1 0 X X
7	0 0	0 1 1 1 X X
8	0 0	1 0 0 0 X X

Tag (AT) ↓ If matches and is valid then it's a HIT



LECTURE 33

27/03

Consider what happens when the processor makes a read request at MM addr A

↳ Cache looks inside itself to see if $tag(A_L) = A_T$

↳ If yes, and $V(A_L) = 1$, then cache has a HIT

↳ Then the cache sends that word quickly to the processor

↳ Otherwise it's a MISS

↳ If the line in A_L is valid, it must be evicted like direct-mapped cache only allows an MM block in one place

↳ The new block is slowly retrieved from MM and copied into cache line A_L

↳ Set $tag(A_L) = A_T$, $V(A_L) = 1$, $D(A_L) = 1$

Eviction of a cache line

- Method 1 → Write-Back Cache

↳ If there was a write/store to any word in the line, the Dirty bit will be set

↳ When $D(A_L) = 1$, upon eviction, the line must be written-back so MM is correct

★ ↳ Multiple writes to a line results in only 1 write to MM

- Method 2 -> Write-Through Cache
  - ↳ Has no Dirty bit
  - ↳ All writes to a line are also automatically sent to MM
  - ↳ Generally undesirable as we may have multiple writes to MM in very short span.

LECTURE 34 31/03

New Cache Type: Associative Caches

- Every MM block can map into every cache line
  - ↳ Good: Solves the conflict problem in best way possible
  - ↳ Bad: Needs a lot of digital circuitry (comparators) to simultaneously check all tags of all lines
- There is a compromise b/w the 2 (direct-mapped & fully associative).
  - ↳ Allows each MM block to placed in a smaller set of lines
  - ↳ Say "N" different lines -> called N-Way Set-Associative Cache"

E.g. 1 → A 2-Way Set-Associative Cache allows an MM block to be placed in 2 different cache lines

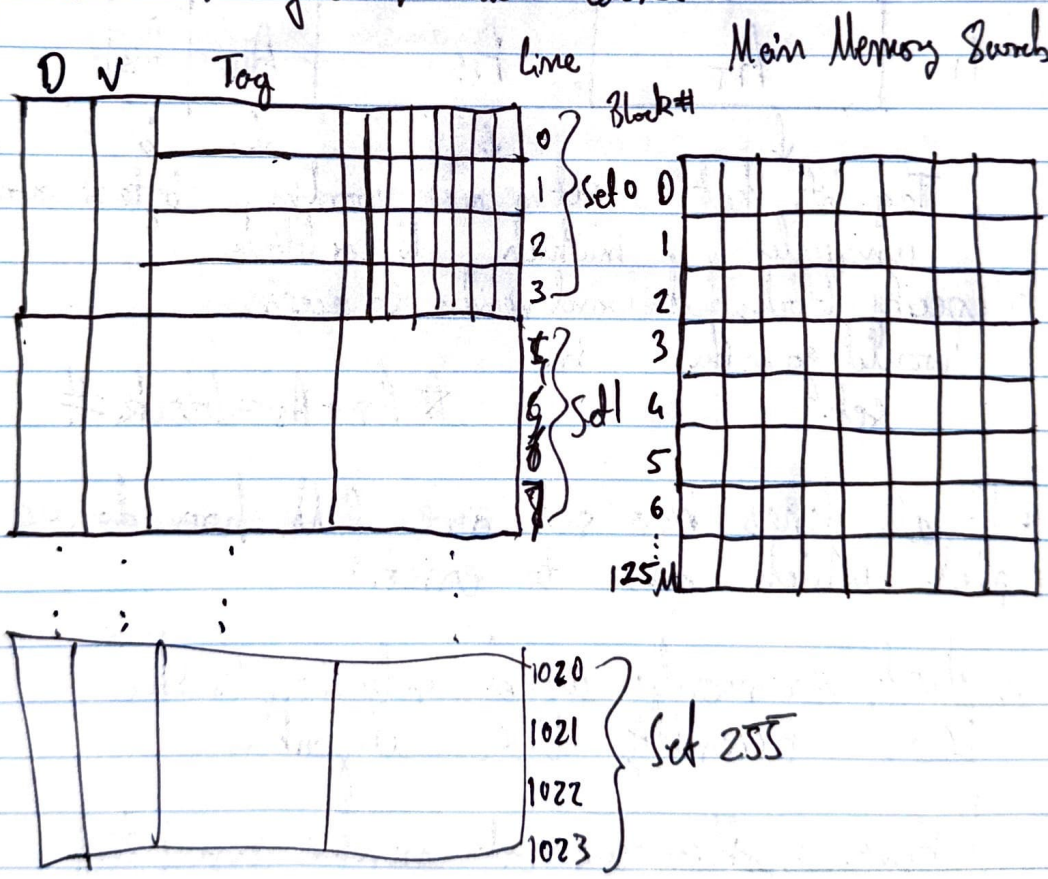
↳ Those 2 lines are called a set.

E.g. 2 → For NIOS-V (addressability of 1 byte), cache size is 32 KBytes

↳ Cache has 32 bytes per line, 8 words per line

↳ It is a 4-Way Set-Associative Cache

* 32K/64K always means nearest power of 2 *



* For an N-Way Set-Associative Cache, each memory block can be placed into N-different cache lines

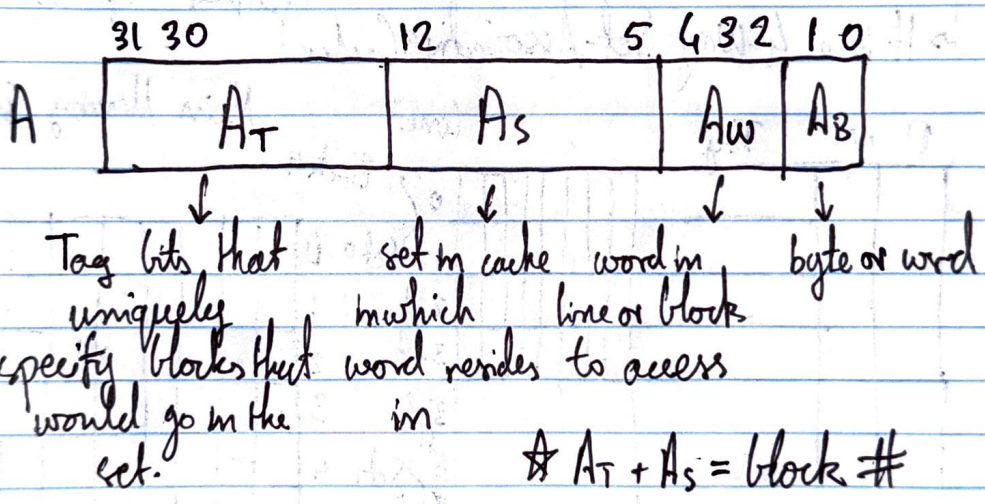
To determine the set-associative cache mapping / HIT/MISS, we need to compute:

↳ # of lines = $\frac{\text{\# bytes in cache}}{\text{\# bytes/line}} = \frac{32K}{32} = 1K = \underline{1024}$

↳ # of sets = $\frac{\text{\# lines}}{N} = \frac{1024}{4} = 256$

↳ Need $\log_2(256) = 8$ bits to select a set

↳ 8 words/line → $\log_2(8) = 3$ bits to determine which word in a line/block to address



☆ $A_T + A_S$ = block #

- If all lines in a set are full, how do we pick which one to evict.

↳ Most commonly used solution is the Least Recently Used algorithm

↳ Have another two bit counter associated w/ each line in the set

↳ Set to 0x00 if line is accessed and increment by 1 each time not used

Choosing randomly in general about same as LRU

LECTURE 35 02/04

- Spent some time on cache simulators → check notes

Analyzing a Program's Cache Behaviour

- Consider some code examples wrt cache behaviour

E.g 1 → Conflicting access (assume DM w/ BS=8)

```
int A[100], B[100], sum;
sum = 0;
for (int i=0; i<100; i++) sum += A[i] + B[i];
```

↳ Suppose every element A[i] maps to same line as B[i]

↳ Accessing B[i] after accessing A[i] will kick 8 integers out of cache

↳ Happens each iteration of for loop, destroys cache

↳ Soln: maybe split into two loops

E.g 2 → Accessing 2D arrays e.g frame buffer from lab 7

```
void bad_clear(void) {
    for (int c=0; c<320; c++) {
        for (int r=0; r<240; r++) {
            but[r][c] = 0x0;
        }
    }
}
```

↳ Horrendous because of row-major array interferes w/ column-major clear function

Eg 3 → Accounting for cache size

```
int A[N], sum, prod;
for (int i=0; i<N; i++) sum = sum + A[i];
for (int i=0; i<N; i++) prod = prod * A[i];
```

↳ If N >> cache size, we have to clear cache and redo whole cache retrieval for the second loop

↳ Solution: do both sum & prod calculations in same for loop

LECTURE 36 03/04

Performance Analysis of Caches Running a Program

- Recall how memory calls work

↳ Processor emits a request to access memory

↳ If memory address in the cache, then we have HIT and responds quickly, say in "C" cycles

↳ These days C ≈ 4 cycles

↳ If cache MISS, takes M cycles for memory to find that address

↳ M is called the miss penalty, ≈ 300 cycles

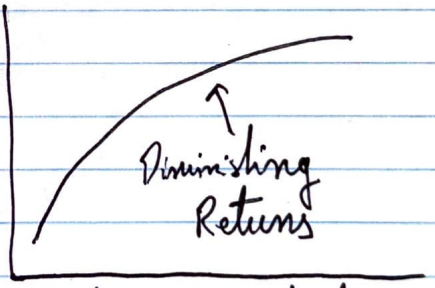
- Define hit rate = num cache HITs / total num accesses
  - always $0 \leq h < 1$
  - We can formulate h using $h, C, \& M$ compute average number of cycles to complete a memory access request.

Eg. How many cycles would 1000 accesses take?

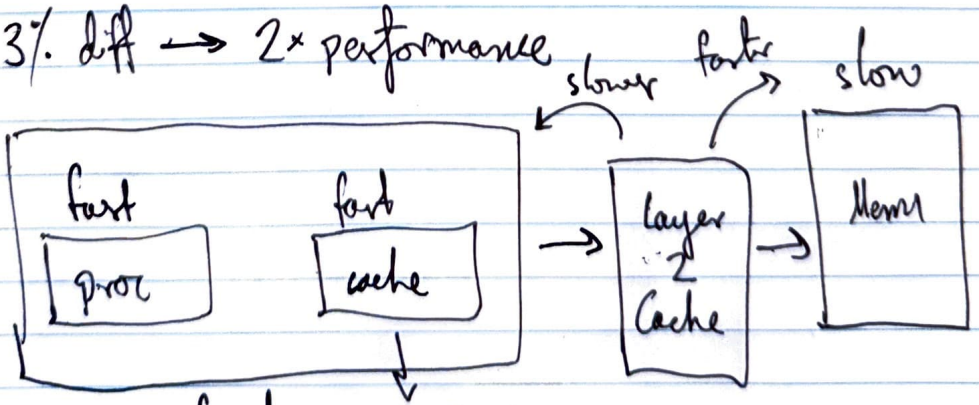
- $\frac{1000 \cdot h \cdot C + 1000 \cdot (1-h) \cdot M}{1000}$

- Have quantity Average Access Cycles (AAC)
  - $h \times C + (1-h) \times M$

h	AAC	
0.85	48	What we want
0.90	34	
0.95	19	
0.98	9.9	
0.99	7.0	



- 3% diff → 2× performance



- Effective miss penalty seen by L1 cache is lower

- Because instruction memory accesses have different patterns that data accesses, the L1 cache is split into
  - L1 instruction cache
  - L1 data cache

  } Architect can design these differently depending on requirements

- Discourse on Intel's reign and arrival of competition
- Discourse on China-Taiwan-TSMC relations