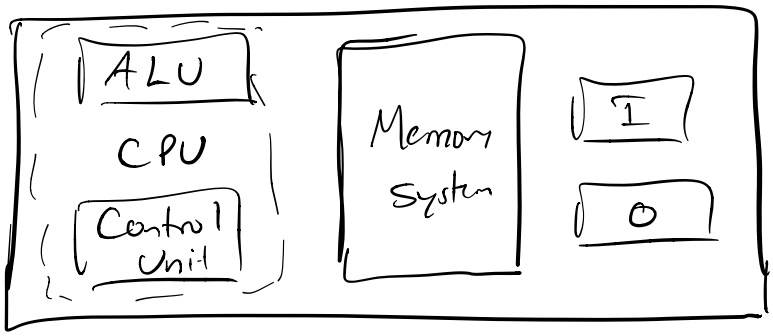
ECE 222

Basics Of Computer Architecture



Program → Assembly / Instruction Set (IS) → Machine Language

Program
- C++
- C
- Java

Assembly / Instruction Set (IS)
- Processor-specific language
- x86, ARM

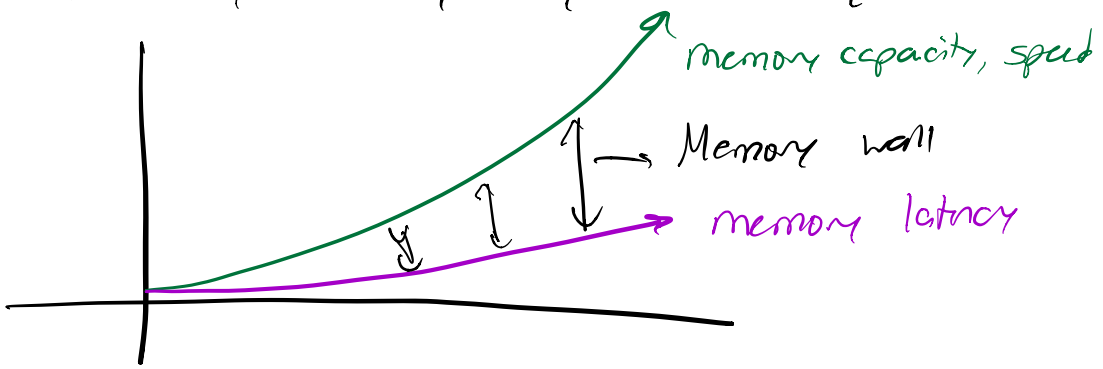
Machine Language
- Seq. of bytes that tell processor to do things
- 10111. . . .

program → OS → compiled → IS → microarch → registr → circuits

- Difference between processor arch. + microarch:
  - Processor arch: interface of hardware to software
  - Microarchitecture: implementation of hardware

Metrics

- Moore's Law: every 1.5-2 yrs, 2x transistors on chip area.
  - Processing speed ↑, energy consumption ↓, memory capacity ↑
  - Memory wall: memory latency only increases by 1.1x



- More and more caches.

- Performance metrics:

Performance → $\frac{\#\text{ of sec}}{\text{task}}$ ⇒ latency (low latency = better performance)

Performance → $\frac{\#\text{ of task}}{\text{sec}}$ ⇒ throughput (his tp = better perf.)

- Comparision of Ferarri w/ bus.
- Comparison: speedup

Speedup of A over B = $N = \frac{\text{throughput}_A}{\text{throughput}_B}$

OR

$N = \frac{\text{latency}_B}{\text{latency}_A}$ ⇒ $N \propto \frac{1}{\text{latency}}$

- Iron Law of Performance: Whole programs

$$\text{CPU time} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{cycle}}$$

(cycles: clock)

- $\frac{\text{instructions}}{\text{program}}$: Compiler, Algorithm, IS
- $\frac{\text{cycles}}{\text{instruction}}$: IS, Microarch
- $\frac{\text{seconds}}{\text{cycle}}$: Circuit, Transistor, Microarch

- Latency: $\frac{\text{seconds}}{\text{instruction}} = \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{cycle}}$
- Throughput: $\frac{\text{instruction}}{\text{second}} = \frac{\text{instruct.}}{\text{cycle}} \times \frac{\text{cycle}}{\text{seconds}}$

↳ MIPS: millions of instr/second.

Ex:// CPI: 2. Clock = 500 MHz ($\frac{cycles}{sec}$). Find MIPS

$$MIPS = \frac{instr}{sec} \times 10^6 \Rightarrow \frac{instr}{2\,cycles} \times \frac{500 \times 10^6\ cycle}{second}$$

$$= \frac{250\ instr}{sec} \times 10^6$$

$$= 250\ MIPS$$

- Amdahl's Law: improving part of the program ⇒ overall speedup?

$$\text{Overall speedup} = \frac{1}{(1 - Frac_{ENH}) + \frac{Frac_{ENH}}{Speedup_{ENH}}}$$

- $Frac_{ENH}$ = proportion of program improved (% of execution time)
- $Speedup_{ENH}$ = how fast did proportion speedup.

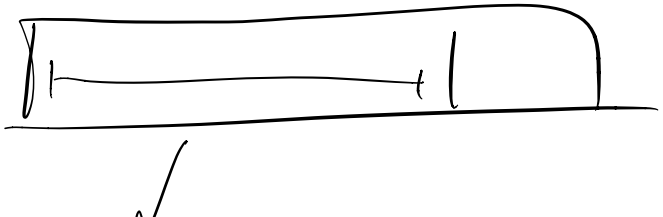
Ex:// 2 improvements for program

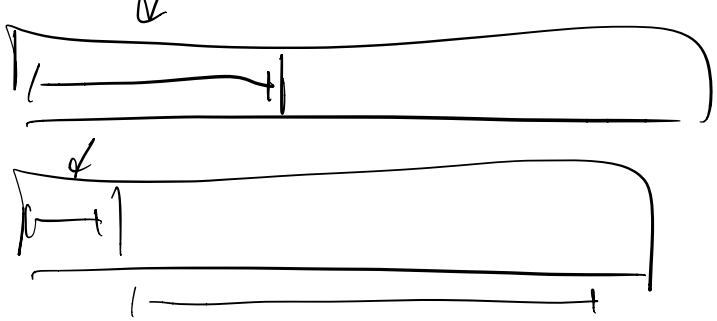
1) Speedup 20x on 10% of total time
2) Speedup 1.6x on 80% of time

1) $Speedup_{overall} = \frac{1}{0.9 + \frac{0.1}{20}} = 1.05$

2) $Speedup_{overall} = \frac{1}{0.2 + \frac{0.8}{1.6}} = 1.43$

- Common improvement > high improvement on small program
- Law of Diminishing Returns: improve multiple aspects of program.





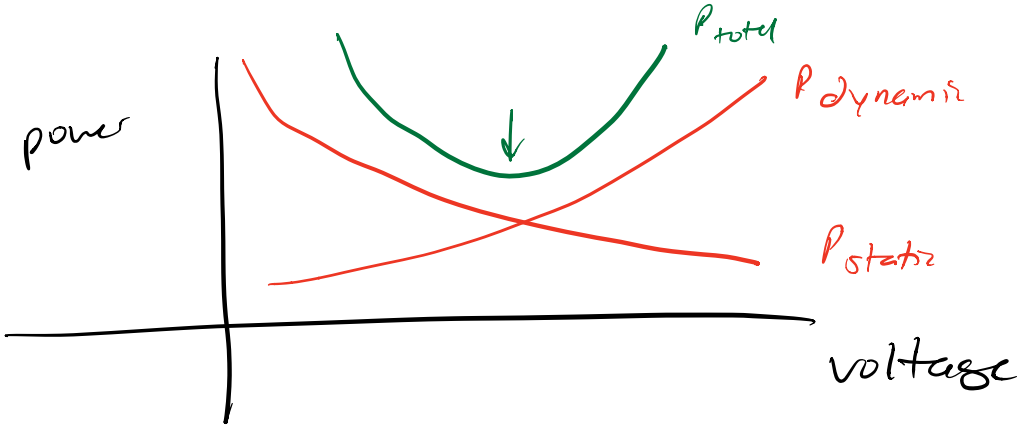
- Power improvements:
  - Dynamic power: activity consumes energy.

$$\text{Dynamic power} = \tfrac{1}{2} C V^2 f \alpha$$

(C: cap. of wire; V: volt.; f: freq.; α: % of active transistor)

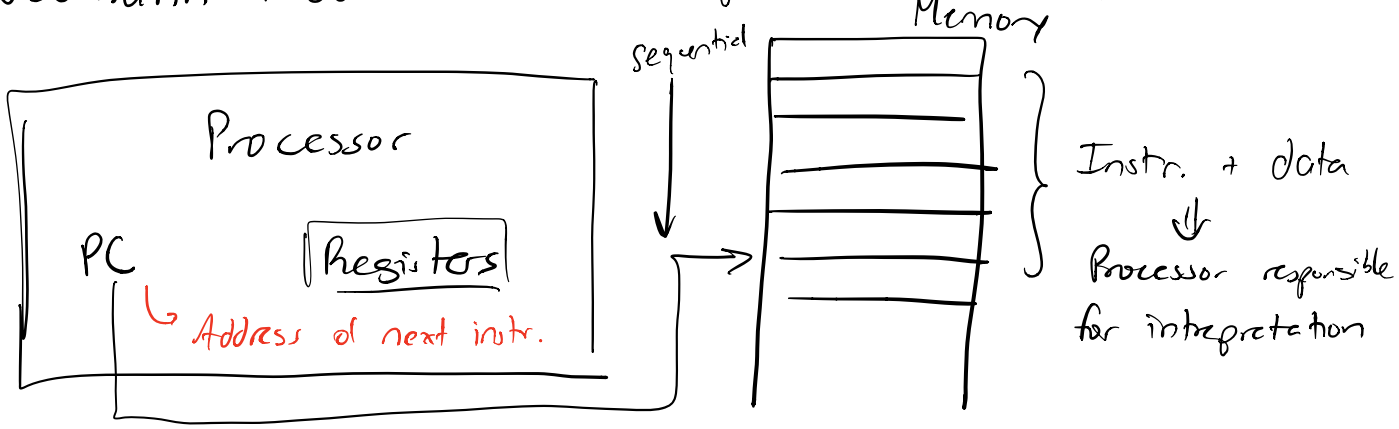
  - State power: idle consumption.

voltage ↓ → leakage ↑ (power wall)



INSTRUCTION SET ARCHITECTURES

- Defn: contract between software + hardware
- Von Neumann: model of all computers. Stored program model



Design of ISAs

RISC (ARM)
- Simple instructions
- $\frac{\text{instr}}{\text{program}} \uparrow$, $\frac{\text{cycles}}{\text{instr}} \downarrow$

CISC (x86)
- Complex instruction ⇒ desc
- $\frac{\text{instr}}{\text{program}} \downarrow$, $\frac{\text{cycles}}{\text{instr}} \uparrow$

Code

- Instruction format: length + encoding of instruction

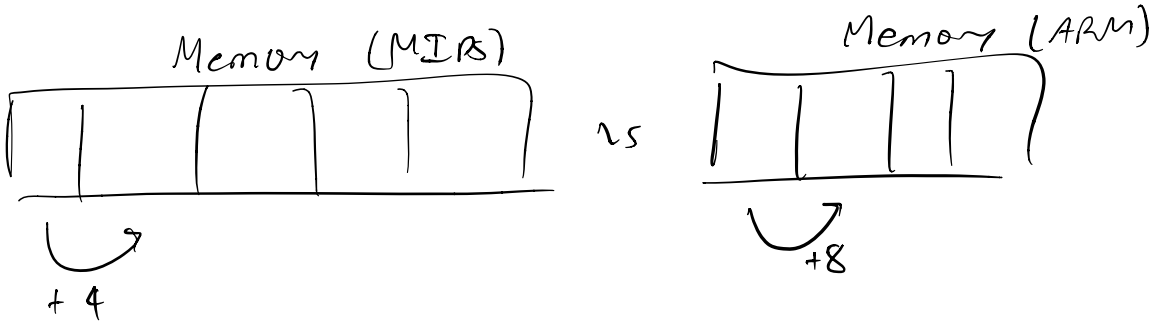
Length:
- Fixed: easy computation but difficult to program
- Variable: opposite pros + cons

Encoding
- # of ways to encode op.
- Simple + low # of encodings

- Opcode: tells us which operation to perform
  - Operand model: # of operands, sequence, types of operands (memory / register operands)

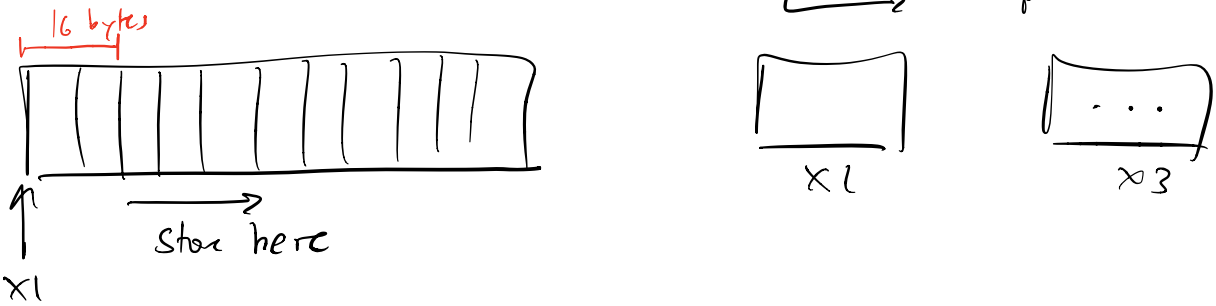
ARM ISA OVERVIEW

- 64-bit architecture: every memory address is 64 bits



- Registers: X0 — X30 (64 bit)
  - Special: XZR (zero register), SP (stack ptr)
  - 32-bit registers: W0 - W30, WZR, WSP
- Labels: refer to specific lines ⇒ "alpha:"
  - Associated w/ line #
- Load + store from memory: LDR/STR destReg, [addrReg, offset]

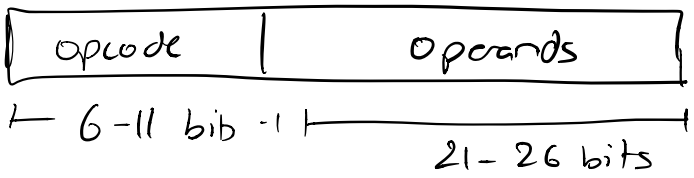
LDR X3, [X1, # 16] → Multiple of 8!



- Arithmetic: operation destReg, Reg1, Reg2 ⇒ destReg = Reg1 op Reg2
- Branching: PC → place in code to execute operation

B ← label, offset

- Machine code converting:



ISA → Assembler → Binary

- Variations: R, I, D, B, CB, IW
- Memory offset:

Offset

Register indirect
LOAD X1, [X2]
↳ X1 = mem [X2]

Displacement
LOAD X1, [X2 + 8]
↳ X1 = mem [X2 + 8]

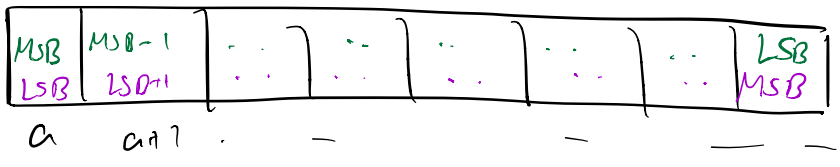
Index base
LOAD X1, [X2, X3]
X1 = mem [X2 + X3]
= mem [i]

Auto increment
LOAD X1, (X2)+
↳ Mem [X2 ++]

Memory indirect
LOAD X1, @ X2
↳ ptr
mem [mem [X2]]
↓ not ARM supported

- Autoincrement in ARM:
  - Pre index: LDR X1, [X2, # 8] ! ⇒ increment X2 after access
  - Post index: LDR X1, [X2], # 8 ⇒ access X2 → increment
- Endianess:

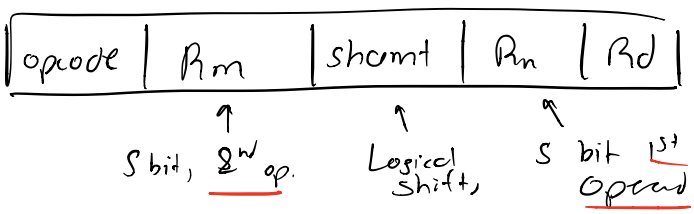
Big Endianess, Small endianess



- Load-store architecture: register – register operations
  - Opposition x86: register-memory

ARM ASSEMBLY

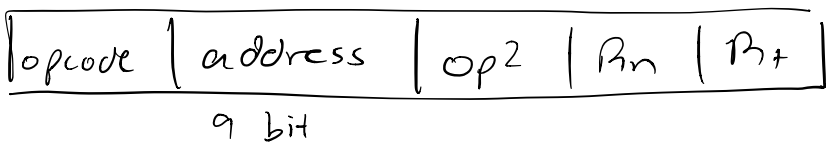
- R format:



  - Arithmetic or logical instructions
  - Logical: LSL, LSR, AND/ANDI, ORR, XOR
    - Ex: LSL X1, X2, 4 ⇒ X1 = X2 << 4
      = X2 * 2^4
      AND X1, X2, X3 ⇒ X1 = X2 & X3
    - Tip: use AND w/ 1 in order to isolate certain bits

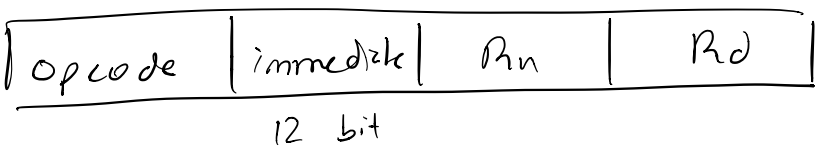
```
  00110
& 00100  ⇒ MASK
-------
  00100
```

- D format:

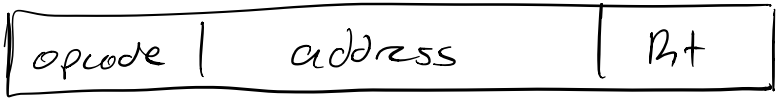


  - Memory access (Load/Store)
  - Address is scaled by data size
  - Aligned:
    - 32 bit ⇒ addr % 4 == 0, 64 bit ⇒ addr % 8 == 0
    - ARM supports aligned + unaligned
      - Aligned: LDR/SDR ⇒ offset is multiple of 8
      - Unaligned: LDUR / SDUR ⇒ ¬ "

- I format: immediate instructions (ADDI, SUBI)



- CB format: branching



19 bit, PC-relative

- ARM instructions: CBZ X1, label (go to label if X1 == 0), CBNZ X1, Label
- Use inequality:

N	Z	V	C
(negative)	(zero)	(overflow)	(carry)

① Set flags in operation

ADDS, SUBS, ...

② Branch on conditions

B.EQ, B.GE, B.GT, B.NZ . . . Label

- Ex: // ADDI W1, WZR, 43
  ADDI W2, WZR, 2
  ADDI W3, WZR, 0
  SUBS W4, W1, W2
  B.GT label => If W1 > W2 => label

- B format:

26 bit ↳ Offset to PC

- Unconditional branches to label (B address)
- Address is in terms of words (32 bits)
  - Represent 2^{26} words => 256 MB (+/- 128 MB)
  - Farther than 128 MB => BR address (not an offset)

- IW format:

2-bit | 16 bits | 5 bits

↳ Bigger than I format

- Used for MOV instructions

MOVZ X1, 255, LSL, 0

Translation: 255 → X1, zero out rest of the bits

- Shift amount: used to move immediate to particular location

Shift immediate by n bits

  - Aligned: 0, 16, 32, 48
- MOVK: does not zero out rest of bits
- Moving more than 16 bits:

Moving > 16 bits

Use MOVZ + MOVK multiple times

Pseudo instruction (converted to another ARM instruction w/ direct binary map)

MOV X1, X2, ...

PROGRAMMING PARADIGMS

- Array access:

C code:

a = A[8] + b ⇒

ARM: (assume addr of A is X3)

LDR X4, [X3, #64] ⇒ A[8] w/ offset in bytes

ADD X1, X2, X4 ⇒ add step

C code:

A[12] = b + A[8] ⇒

ARM:

LDR X4, [X3, #64]

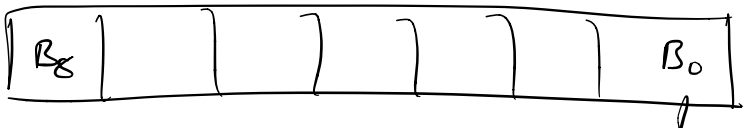
ADD X1, X2, X4

STR X1, [X3, #96] ⇒ A[12]

- Assumption: each item in array uses 8 bytes. Known beforehand

- Move/store bytes:

LDRB / STRB targetReg [addrReg, offset]



↳ Operated on first

- Sign extend: dealing w/ bytes loses sign information
  LDRSB, STRSB (doesn't matter if unsigned)

- If-then-else:

C code:

```
if (a < b) a++;
else b++;
```

ARM (x1 = a, x2 = b)

```
SUBS  X3, X1, X2
B.GE  else
ADDI  X1, X1, 1
B     join   ← skip else!
else: ADDI  X2, X2, 1
join: ...
```

SUBS, B.GE → Condition
ADDI, B → a ≤ b
ADDI → a > b

- For loops:

C code

```
long int A[100], sum, n, i;
for (i = 0; i < 10; i++) {
    sum += A[i]
```

ARM (x1 = i, x2 = N, x3 = sum[], x4 = sum)

```
MOVZ  X1, 0, LSL 0
loop: SUBS X5, X1, X2
      B.GE exit
      LSL  X6, X1, 3
      LDR  X7, [X8, X6]
      ADD  X4, X4, X7
      ADDI X1, X1, 1
      B    loop
exit: ...
```

SUBS, B.GE → cond check
i should access ⇒ LSL X6, X1, 3
array ⇒ byte
access ⇒ i * 8
LDR, ADD, ADDI → sum += ...

- Case statement:
  - Using if-else ARM paradigm is really inefficient
  - Use branch table:

```
br_table: label_case_0
          :
          :
          label_case_n
```

Array of double words stored in memory

  - Implementation:

```
ADR  X0, br_table  => br_table absolute addr => X0
LSL  X1, X1, 3     => Offset (set X1 to be numerical case, 5, 6...)
LDR  X0, [X0, X1]  => Collecting label
BR   X0            => Branching on label
```

- Bound checking:
  - Out-of-bounds: lower then min or high then max

Options

B.GT & B.LE

i as unsigned + use B.GE
(really negative i => unsigned really big)

```
SUBS  XZR, X1, X2
B.HS  outofbounds
```

PROCEDURES

Caller → calls → Callee => Function

Problems

Passing control | Passing values | Preserving registers

- Passing control:

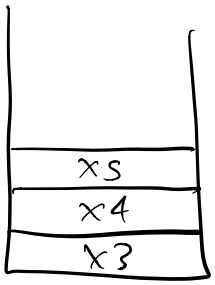
```
main: . . .
      . . .
      BL foo   → 1. PC set to foo
                 2. Previous PC stored in LR
foo:  . . . .
      . . . .
      BR LR    → [X30] tells program to go to stored PC
```

- Passing values:
  - Soln: designate static special registers

- Pass by value (give values) / pass by reference (give addr)

- Preserving registers:

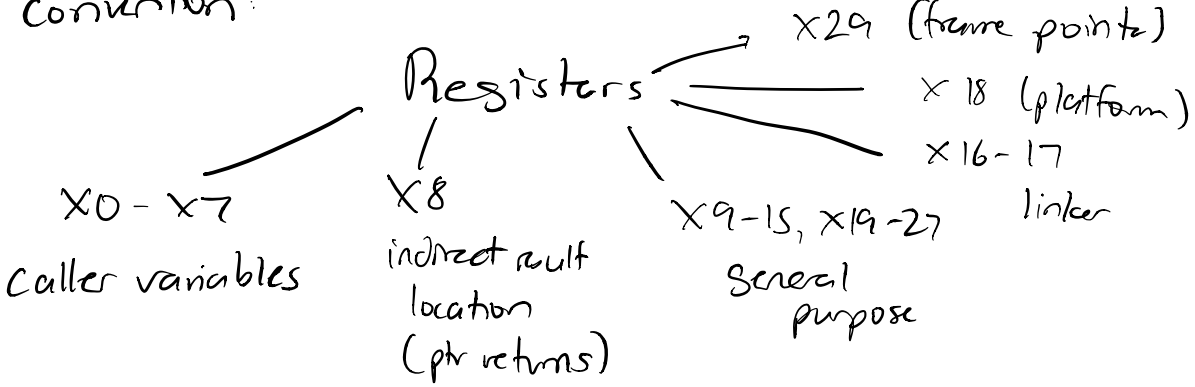
Stack: saving register s.t. we can restore later



- Formula:

```
foo:  SUBI  SP, SP, #(8*n)   } Decreasing stack ptr + storing
      STR   XN, [SP, #0]     }
      :
      function work
      :
      LDR   XN, [SP, #0]     } Loading + increasing stack ptr
      ADDI  SP, SP, #(8*n)   }
      BR LR
```

- Register convention:

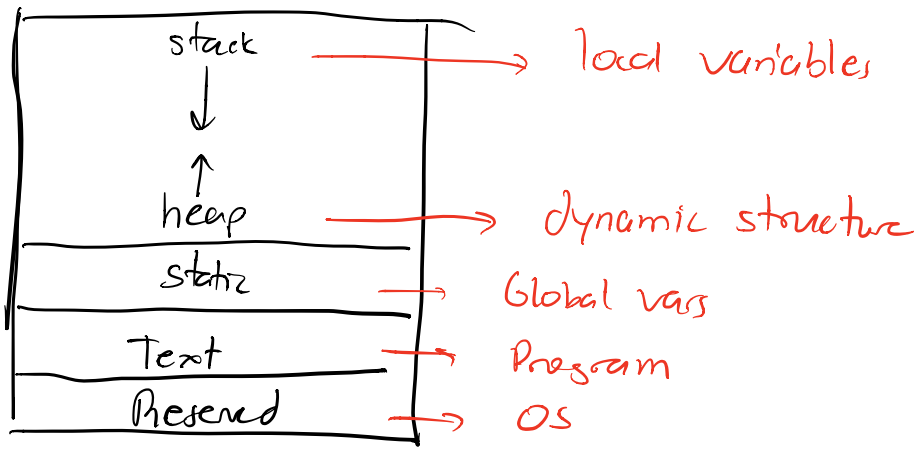


- Procedure of procedures:

1. Pass args to procedures: X0-X7 registers/stack
2. Save caller-save registers + adjust SP
3. BL
4. Establish stack frame: SUBI, SP, SP, #(8*n)
5. Save callee-save registers: variables that callee uses (eg. LR)
6. Work

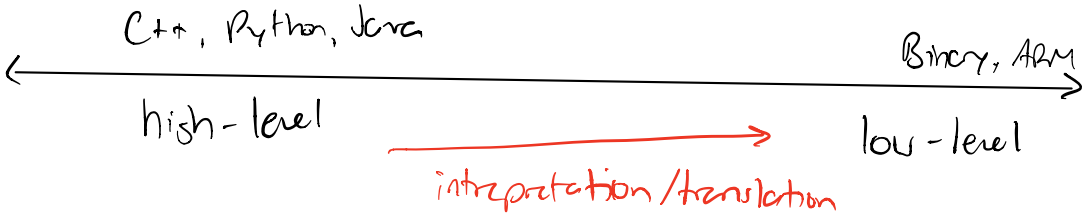
7. Put return value into x0 - x7
8. Restore callee-save registers
9. Pop stack
10. Return to caller

- Structure of memory



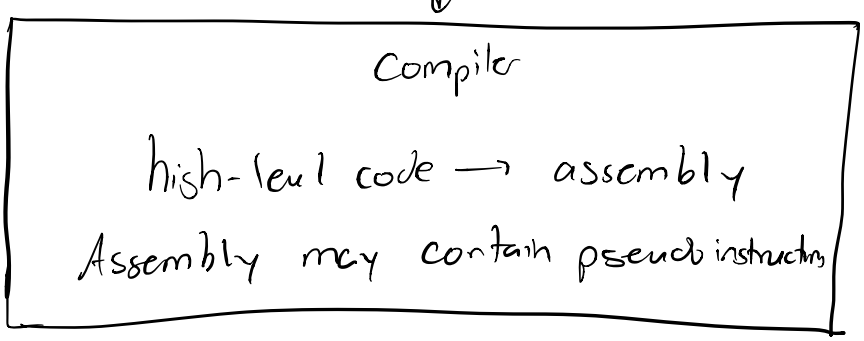
- Character data: use appropriate loading/storing based on bit encodings
  - Ex: 8 bit encoding ⇒ LDRB/STRB

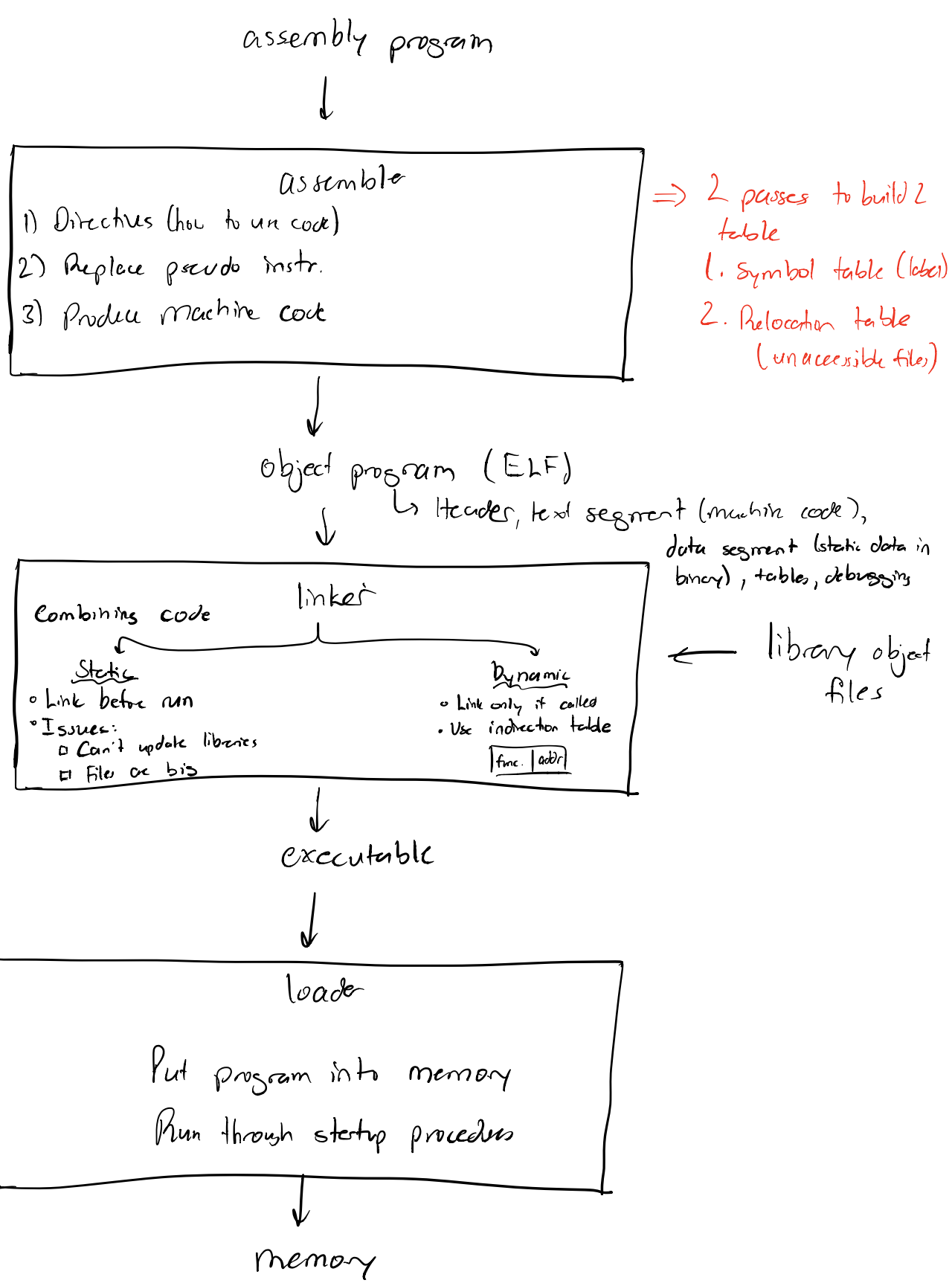
COMPILING, ASSEMBLING, LINKING, LOADING



- Interpretation is quite slow (eg. Python), so we use compilers/translators

program (.c, .cpp...)





PROCESSOR DESIGN: DATAPATH I

CPU

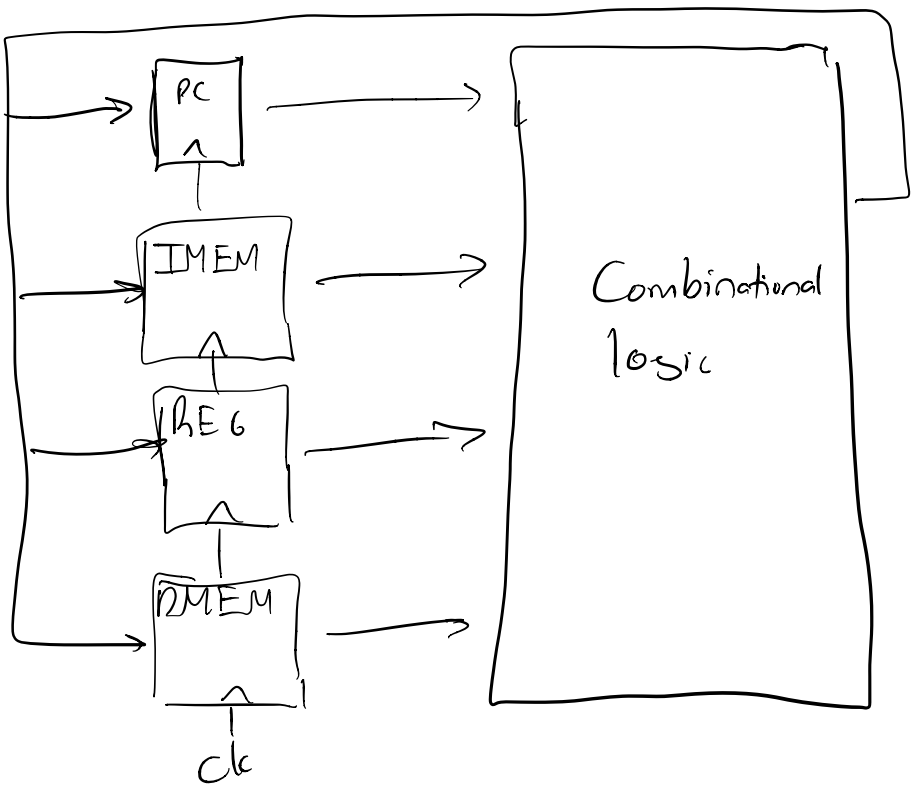
Datapath
- Hardware to perform ops.

Control
- How datapath should work

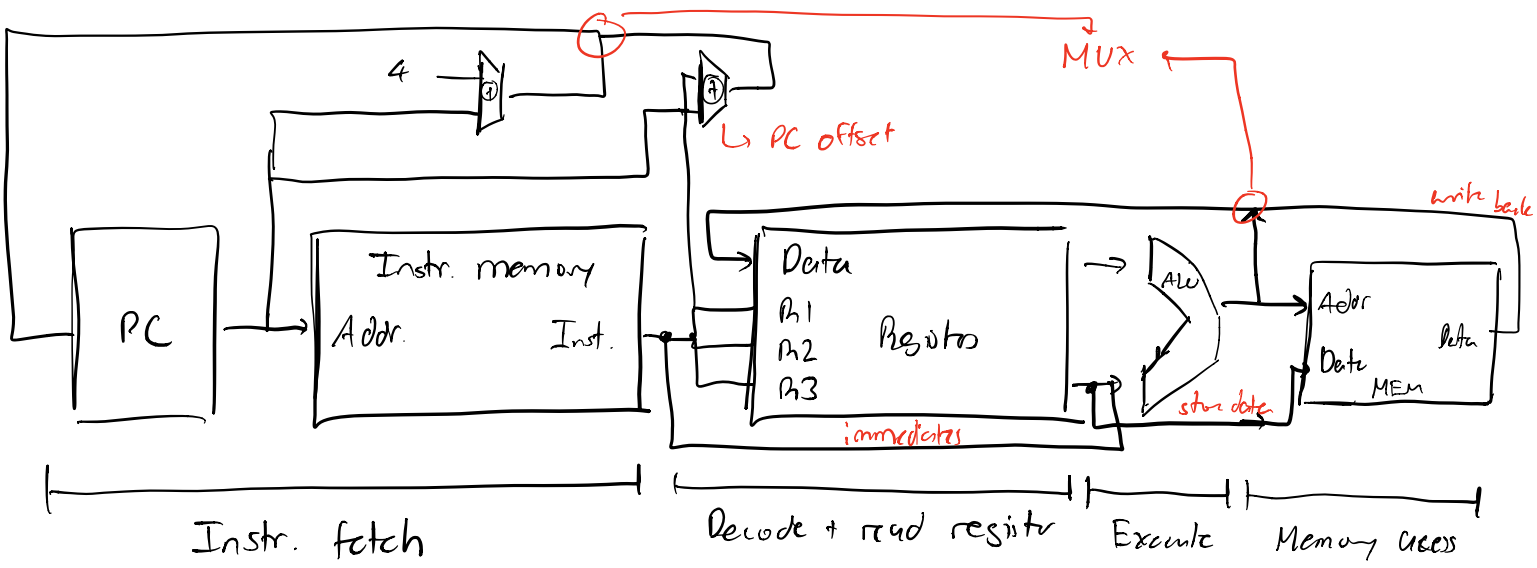
- Designing processors:
  1. Create data path req. based on ISA
     - ISA model → register transfer model (RTL)
  2. Select components + choose clocking patterns
  3. Assemble components to meet RTL model
  4. Analyze ISA implementations + determine control points
  5. Assemble circuit logic for control points

  Iterate until good

- Processor design impacts CPI & clock cycle speed
- Single-cycle processor: 1 cycle/instruction ⇒ very low clock speed

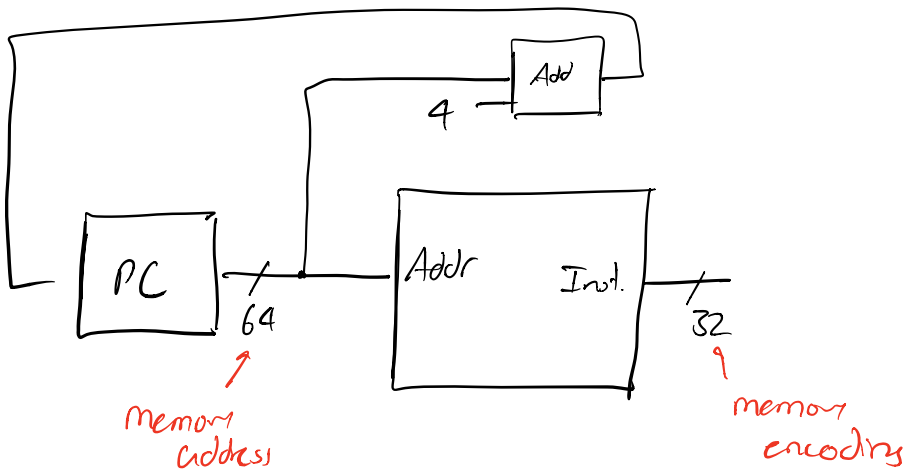


- Datapath stages:
  1. Fetching instructions: puts instr from mem to reg., PC += 8...
  2. Decoding " : read opcode → extract data → reg.
  3. Execute : logical/arithmetic ⇒ ALU. Load/stores: offset arithmetic
  4. Memory access: load/store
  5. Write to reg: results → dest. reg (not done by loads/stores)

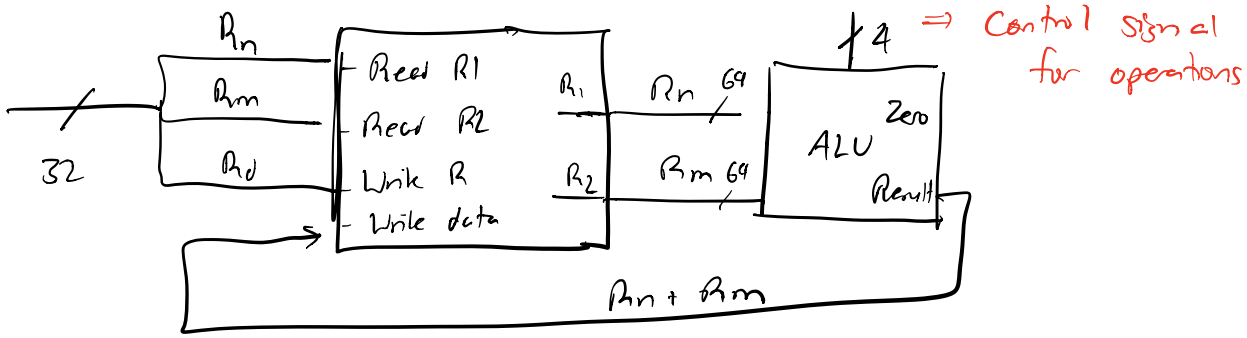


PROCESSOR DESIGN : DATAPATH II

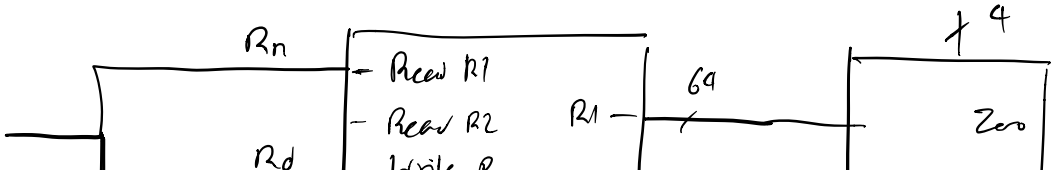
- Need processor to decl w 6 different inst. formats
- Fetch:

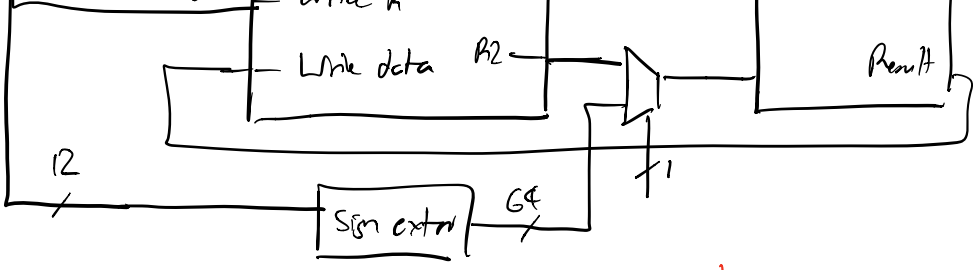


- ADD: R format instructions



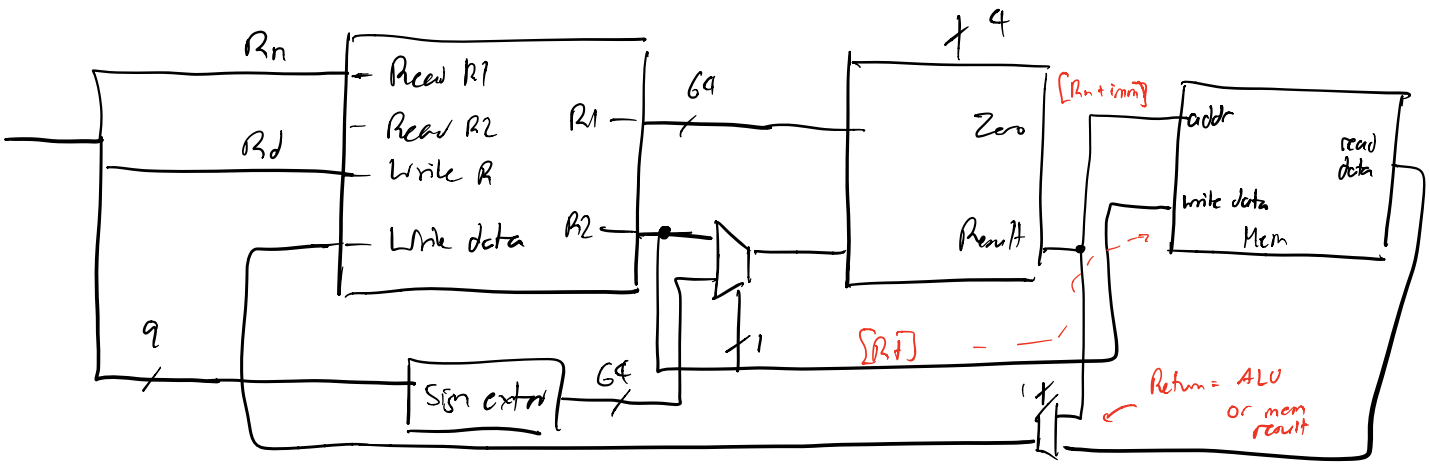
- ADDI: I format



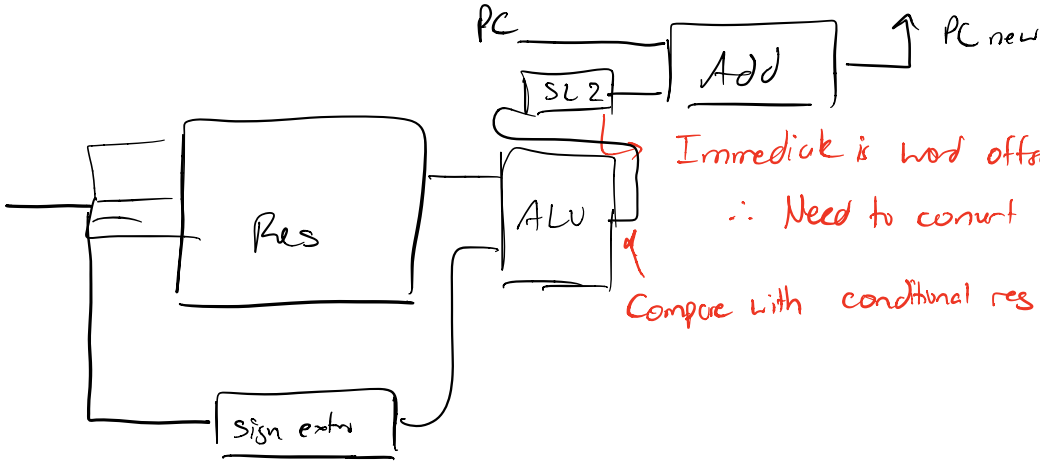


↳ We need to make immediate 12 bits → sign ext.

- LDR/STR: D format



- CBZ: B format



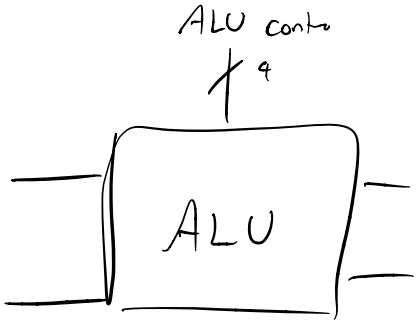
- Conditional flags: set or conditional registers (R[31:28])
- If asked to do datapath for other instructions:
  1. Break down what hardware needs to do via datapath stages
  2. Connect on diagram
- Reasoning behind 5 stages:
  1. Longest instruction can still function
  2. Minimize amount of hardware

PROCESSOR DESIGN: CONTROL

Opcode $\xrightarrow{\text{comb. logic}}$ control signals

- Control signals:
  - ALU control:

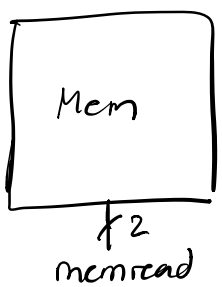
ALU control	function	ARM
0000	AND	AND
0001	OR	ORR
0010	add	ADD / ADDI
0110	sub	SUB / SUBS
0111	pass	1 reg. inputs (CBZ)
1100	NOR	ORN
⋮	⋮	⋮



[!] To generate ALU control: multi-level control gen.

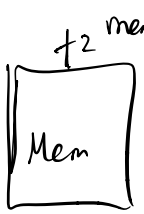
ALU opcode + inst. opcode ⇒ ALU control ⇒ efficient

  - Mem read: do we need to read from memory



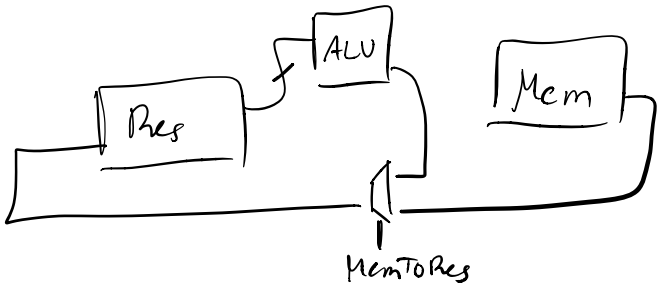
* enablement

  - Mem write: do we need to write data to memory

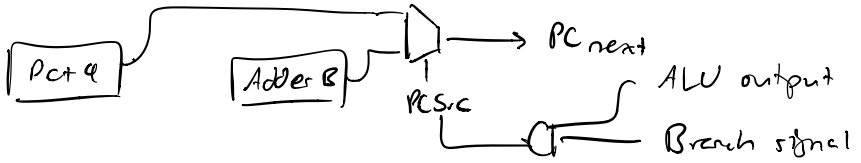


* enablement

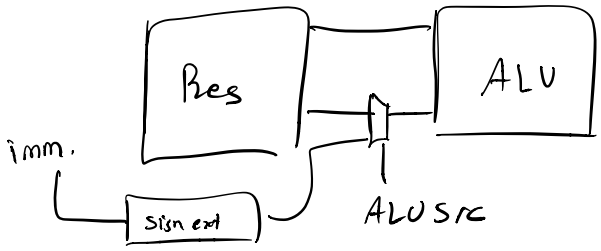
  - MemToReg: do we need to write ALU (0) / Mem (1) data to reg?



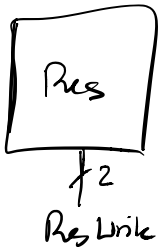
  - PCSrc: should we select normal, PC+4 (0) or branch target (1)



- ALU Src: add w/ register (0) or w/ immediate (1)

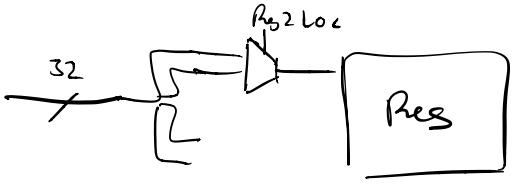


- RegWrite: should we allow write back to reg.



* enablement

- Reg2Loc: select reg from R format [16-21] (0), or select D/CB [4-0], (1)



⇓

x for LDR because only reading 1 register!

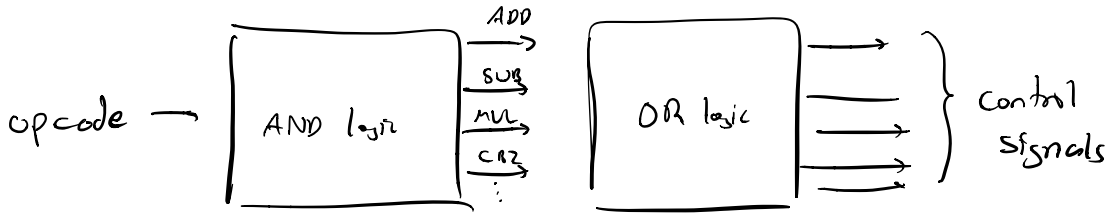
- Strategy to determine control signals for operations
  1. Write down all signals (inc. branch signal)
  2. Think back to defn. of signal ⇒ whether needed or not
  3. If signal is not needed ⇒ X
  4. If it is ⇒ 0, 1, ...

- Ex:// CBZ format. Find control signals

Control Signals	Values
Reg2 Loc	1
Branch	1
MemRead	0
Mem Write	0
Mem to Reg	X
ALU Op	0111
ALUSrc	0
Reg Write	0

- Dealing w/ unconditional branches:
  - No ALU input necessary

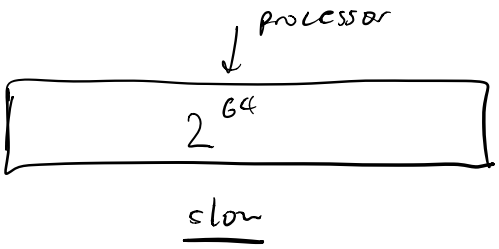
- Implementing control signals
  1. Use ROM to store all control signals for all instruction possibilities
     ↳ Problem: unrealistic for large ISAs
  2. Combinational logic



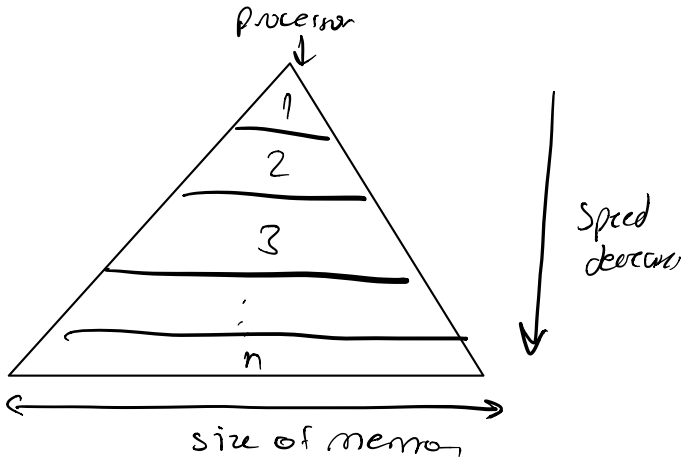
- Hidden efficiency: group similar inst ⇒ similar AND logic

- Overall problem: single-cycle is really slow!! Cycle time = time for longest instruction

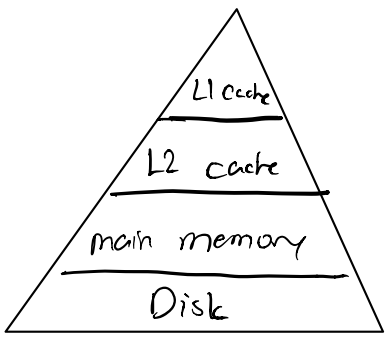
MEMORY HIERARCHY



⇒



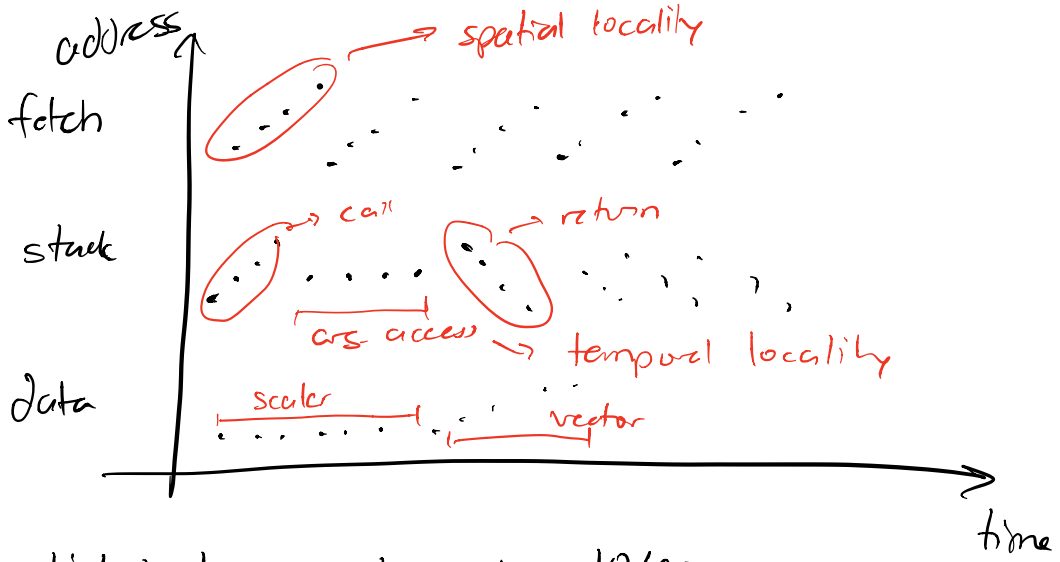
- Implementation:



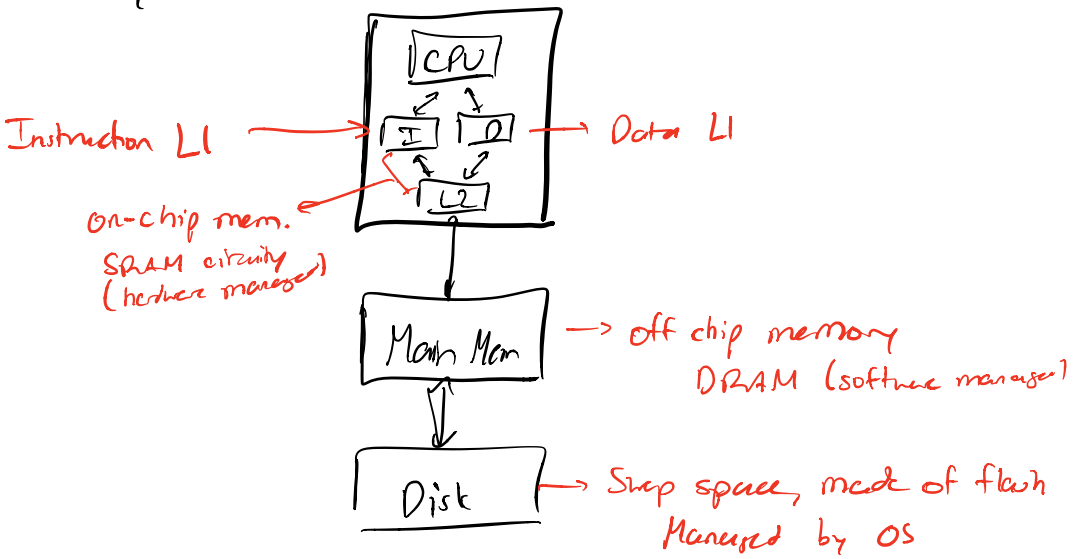
- Reasons why memory hierarchy works:
  1. Temporal locality: recent exec. inst. more likely to be exec. again
     ☐ Can bring inst. into cache ⇒ use it again + again
  2. Spatial locality: data near exec. inst. likely to be exec. as well

- Can bring set of local data/inst. into cache

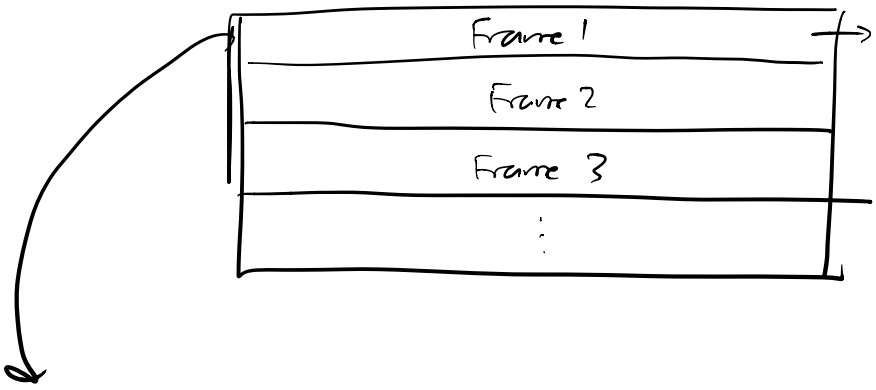
- Patterns in program:



- Reason why spatial + temporal exist: 10/90 rule
  - Instructions: 10% of static inst. account for 90% of dynamic inst.
  - Data: 10% of variables account for 90% of accesses
    - Variables → global.
- Hit: access mem. level + find data
- Miss: access mem. level + don't find data ⇒ move to next mem. level
  - Expensive: requires you to bring additional data back to top level
- Goal of memory arch:
  1. High hit rate / minimize miss rate
  2. Low latency



CACHE ORGANIZATION



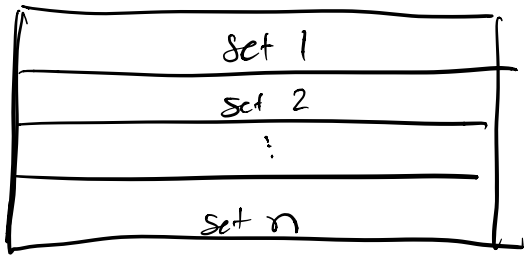
1: valid

0: invalid

label blocks for comparison

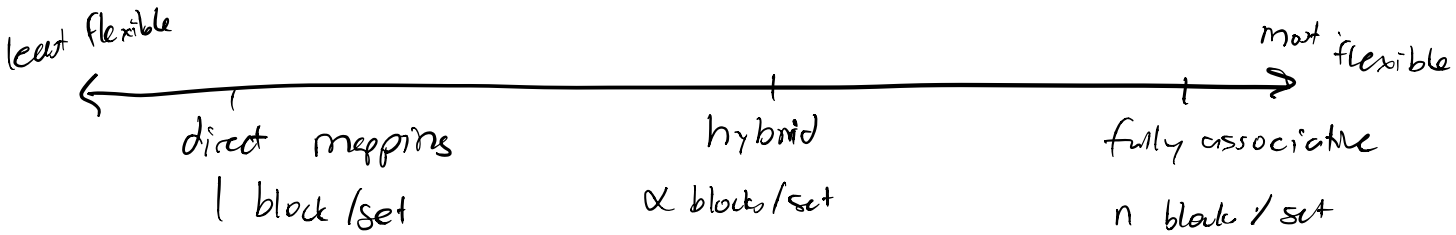
- Like to group frames into sets

data mod n →



Bits of the block act as the valid, tag, set tags

- Each set has a number of frames



- If data maps to already filled set ⇒ replace existing data
- Ex: // 1. Computer turns on

valid	tag	block
0	[0-31]	junk
⋮	⋮	⋮
0		

2. CPU asks for word at address [32-35]
   1. Which block does this belong to?
      Because aligned ⇒ [32-63] block has data
   2. Check cache: miss
   3. Pull from lower level of memory

valid	tag	block
1	[32-63]	. . .
⋮	⋮	⋮
0		

3. CPU asks for word at address [32-35] again
   1. Block? [32-63]
   2. Check cache: hit
4. CPU asks for word at address [36-39]
   1. Block? [32-63]
   2. Check cache: hit

(8B block)

- Ex: // Direct mapped, 8 set cache. Stream of 1 byte requests:

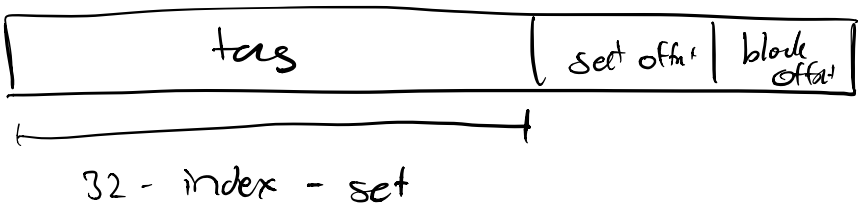
  2, 11, 5, 50, 62, 51, 3

  Sequence of hits + misses.

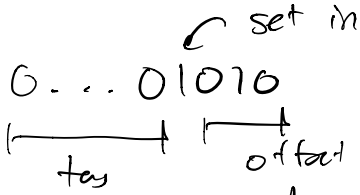
1. 2 → block [0-7] → miss (set 0)
3. 11 → block [8-15] → miss (set 1)
4. 5 → block [0-7] → hit (set 0)
5. 50 → block [48-55] → miss (set 6)
6. 67 → block [64-71] → miss (set 0) ⇒ 67 is not in set 0 !! Set 0 rpl.
7. 51 → block [48-55] → hit (set 6)
8. 3 → block [0-7] → miss (set 0)

- To solve: n-way associative set (n frames/set)

- Mapping address to sets:
  - Block size = N bytes. Need $\log_2 N$ bit to identify bytes
  - Set = S sets. Need $\log_2 S$ bits to identify set
  - Rest of bit: tag



- Ex:// Address 10:



} dependent on architecture

↳ 2nd byte access

- If mapping to set + miss, which frame to replace?
  - LRU (least recently used): find block not used recently + replace
    - Expensive to implement
  - NMRU (not most recently used): any block not used replace (LRU for 2-set associative)
  - Bélády's algo: optimal, but need to know future access

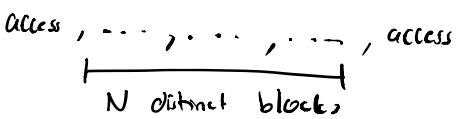
CACHE ORGANIZATION II

Cache design

- **Associativity** (# of frames / set)
- **Block Size** amount of data brought into cache
- **Capacity** amount of data that cache can hold

Reasons for cache misses

- **Compulsory (cold) miss**
  - Cache has never seen data
- **Capacity miss**
  - Seen by cache but kicked ∈ cache too small
  - If N blocks shared by cache:



- **Conflict miss**
  - Not compulsory/capacity
  - Cache associativity is low

- Problem solving strat: classifying a miss
  1. Draw out cache : size of cache, block size
  2. Fill in w/ stream of data
  3. Use properties to figure out miss

- Ex: // 8B blocks. Stream: 2, 11, 5, 50, 67, 121, 256, 512, 1024, 2.
  Last access a conflict / capacity miss?

  2: [0-8)
  11: [8-16)
  5: [0-8) * kicked it out → multiple frames/set; → 1 frame/set
  50: [48-56)
  ⋮

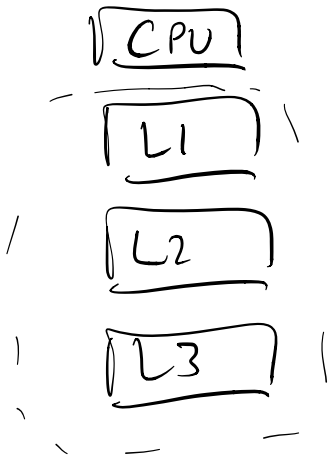
  Only 7 distinct blocks ⇓ Conflict miss.

- Minor optimizations:
  - Associativity ↑: ↓ conflict misses, hit latency ↑
  - Block size ↑: compulsory misses ↓, capacity misses ↓↑, conflict misses ↑
    - Reasoning behind conflict misses ↑: cache size is same ⇒ set # ↓
  - Capacity ↑: capacity misses ↓, hit latency ↑
- Performance metric:
  - Average memory access time:

$$AMAT = t_{hit} + \%_{misses} \times t_{miss}$$

($\%_{misses}$: # of misses / # of accesses; t_{miss}: time to replace block)

- For multiple caches:



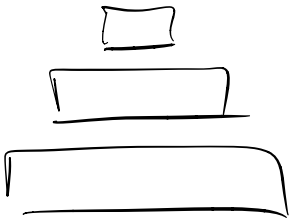
Abstract: $t_{miss-Ln} = AMAT_{Ln+1}$

$$AMAT = t_{hit} + \%_{miss} \times t_{misses}$$
$$= t_{hit-L1} + \%_{miss-L1} \times t_{misses-L1}$$ (recursion!)
$$= t_{hit-L1} + \%_{misses-L1} \times (t_{hit-L2} + \%_{miss-L2} \ldots)$$

Hierarchies

Inclusive:

L_{n+1} is superset of L_n

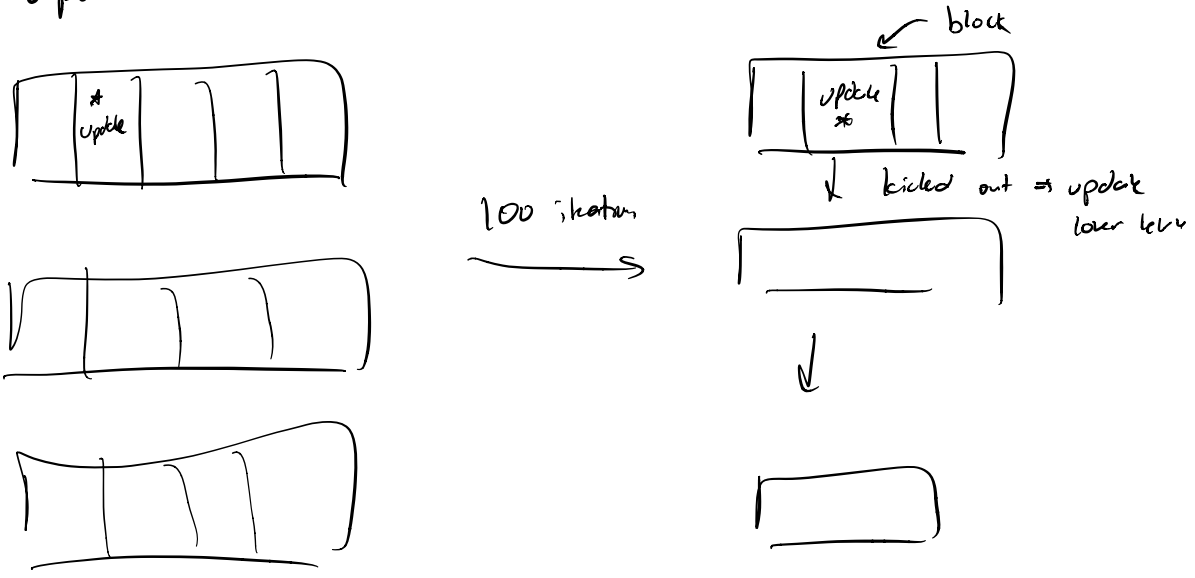


(assume this is true)

Exclusive:

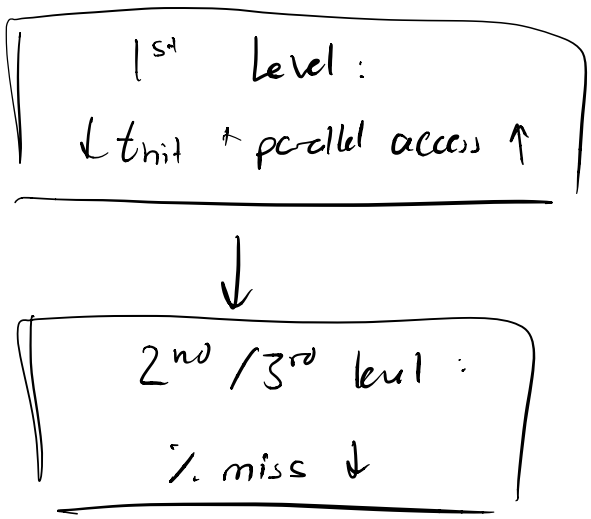
$L_{n+1} \cap L_n = \varnothing$

- How to change data in our cache (write hit)?
  - Method #1: write-through
    - Immediately update lower levels if new value
    - Pros: simple, miss latency is same, cache coherent (caches have same data)
    - Con: time!
  - Method #2: write-back
    - Only update when cache block is going to be kicked out



    - Issue: cache needs to know if block replaced is a modified block
      - Dirty bit: tells cache if modified or not
      - Pros: minimum time
      - Con: non-uniform miss latency (completely dependent on dirty bit!)
- Write miss policies?
  - Method #1: Write-allocate
    - Updates block in memory → update cache
    - Works well with write back
  - Method #1: no write-allocate
    - Update block in memory → no cache update
    - Works well with write through
- Optimization for writing: write buffer
  - If miss ⇒ write to buffer. Will execute later

- Optimization for diff. levels of hierarchy



- Performance metrics in general:
  - Ex:// Stream of data: 20% stores, 80% loads.

    L1: $t_{hit} = 1ns$, % miss = 5% (write through + write buffer)

    L2: $t_{hit} = 10ns$, % miss = 20% (write back, 50% dirty)

    Main: $t_{hit} = 50ns$, % miss = 0%

  1) What is t_{avg} w/out L2:

$$AMAT = t_{avg} = t_{hit} + \%_{miss} \cdot t_{miss}$$

$$= t_{hit\text{-}L1} + (\%_{miss_{load}} + \%_{miss_{store}}) \cdot t_{miss}$$ (store: 0 b/c write buffer)

$$= t_{hit\text{-}L1} + (\%_{load} \cdot \%_{miss}) \cdot t_{miss}$$

$$= 1ns + 0.8 \cdot 0.05 \cdot t_{hit\text{-}main}$$

$$= 1ns + 0.8 \cdot 0.05 \cdot 50ns$$

  2) What is t_{avg} w/ L2:

$$AMAT = t_{avg} = t_{hit} + \%_{miss} \cdot t_{miss}$$

$$= t_{hit\text{-}L1} + (\%_{load} \cdot \%_{miss}) \cdot t_{AMAT\text{-}L2}$$

$$= t_{hit\text{-}L1} + (\%_{load} \cdot \%_{miss}) \cdot (t_{hit\text{-}L2} + \%_{miss\text{-}L2} \cdot t_{miss\text{-}L2})$$

$$= t_{hit\text{-}L1} + (\%_{load} \cdot \%_{miss}) \cdot (t_{hit\text{-}L2} + 2(\%_{mis} \cdot \%_{dirty} \cdot \%_{miss} \cdot t_{hit\ mem}) + (\%_{load} \cdot \%_{miss} \cdot t_{hit\ mem})$$ (2 → dirty miss)

- Tag storage cost:
  - Considered overhead (not included in capacity of cache)
  - Ex:// 4 KB direct-mapped cache. 1024 4B frames.

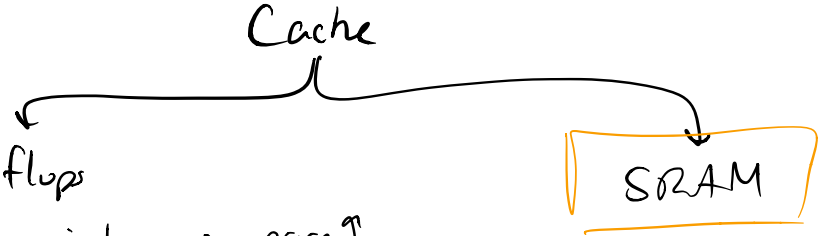
    4B frames: 2 bit offset (block)

    $\frac{4KB}{4B} = 1024$ frames $\Rightarrow$ 1024 frames: 10 bit offset (frame offset)

    Assume 32-bit addr: 20 bits leftover $\Rightarrow$ tag

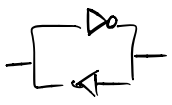
    For all frames: 20 bits × 1024 frames = 2.5 kB
- 2 more optimizations:
  - Victim buffer: used to reduce conflict misses
    - Fully associative cache, but much smaller
    - Block kicked out → VB. Miss → check VB
  - Prefetching: reduce capacity + compulsory miss
    - Anticipate future misses via hardware logic / compiler
    - Next block prefetch: if miss on block X → take X + (X + block_size)
    - Problems: more time + accurate (don't remove useful cache blocks).

CACHE IMPLEMENTATION



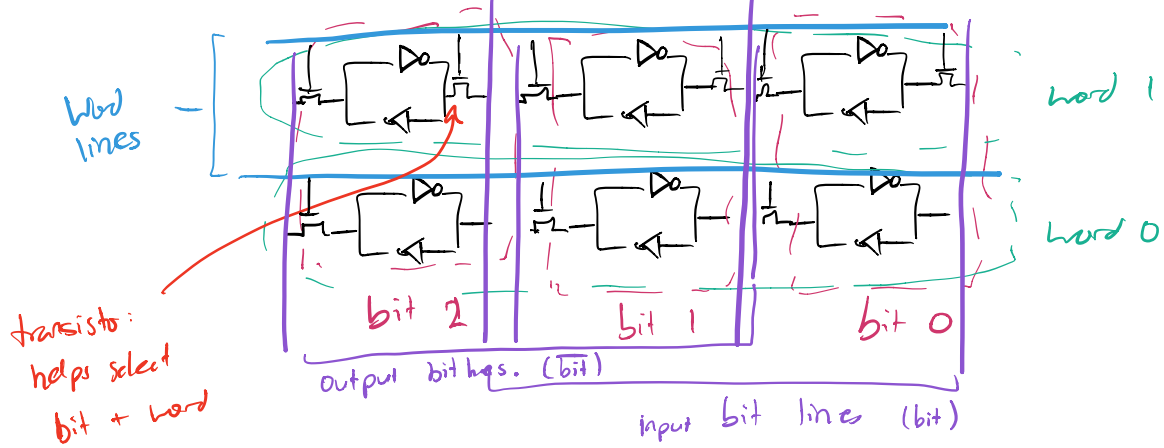
- Lots of transistors $\Rightarrow$ space↑

- SRAM: cross coupled inverter. 4-6 transistors / cell

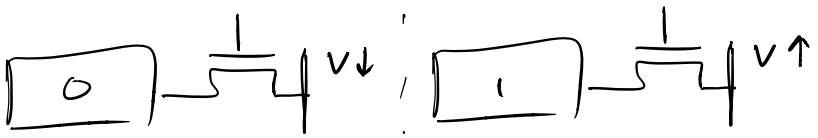


- Static: bit maintains value, no refreshing
- Volatile: power gone → memory gone

- Organization:



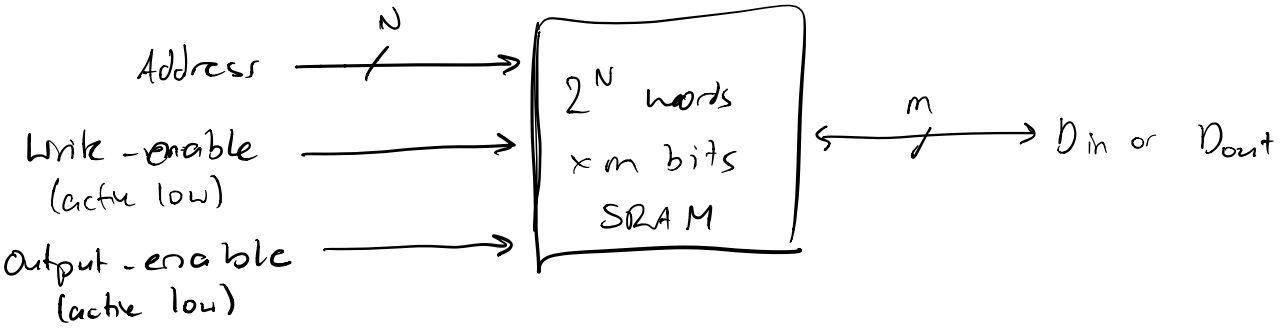
- Reading to SRAM:
  1. Set interested bitlines to 1
  2. Select interest word to 1
  3. Cell will pull bitline up/down.
     - Transistors use voltage differences.



  4. Use sense amplifier to detect difference in voltage (too small to measure) between output + input bitlines.

- Write to SRAM:
  1. Drive bit lines
     - Store 1: bit = 1, $\overline{bit}$ = 0
     - Store 0: bit = 0, $\overline{bit}$ = 1

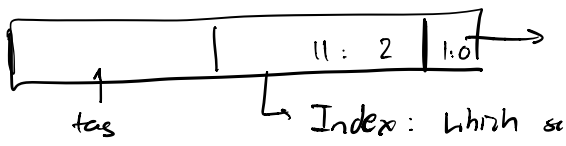
     } $\overline{bit}$ = output bit
  2. Select word via word line
- Length of bitlines >> length of wordlines ← # of words >> # of bits
- Logical diagram of SRAM:



- WE ↓ & OE ↑ ⇒ D = Din
- WE ↑ & OE ↓ ⇒ D = Dout

- Ex:// 4KB cache w/ 1024 4B frames & direct mapped.

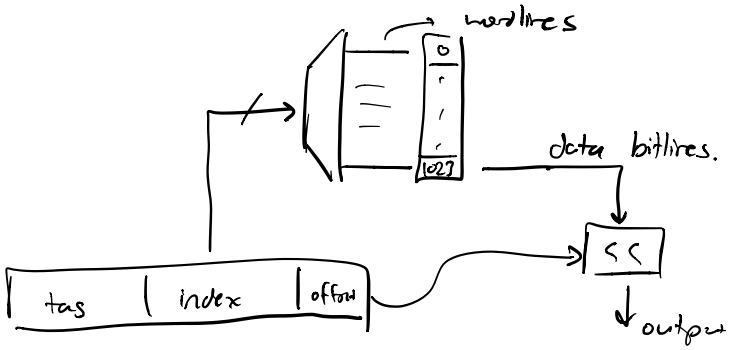
① Dissect input address given problem + 32-bit arch.



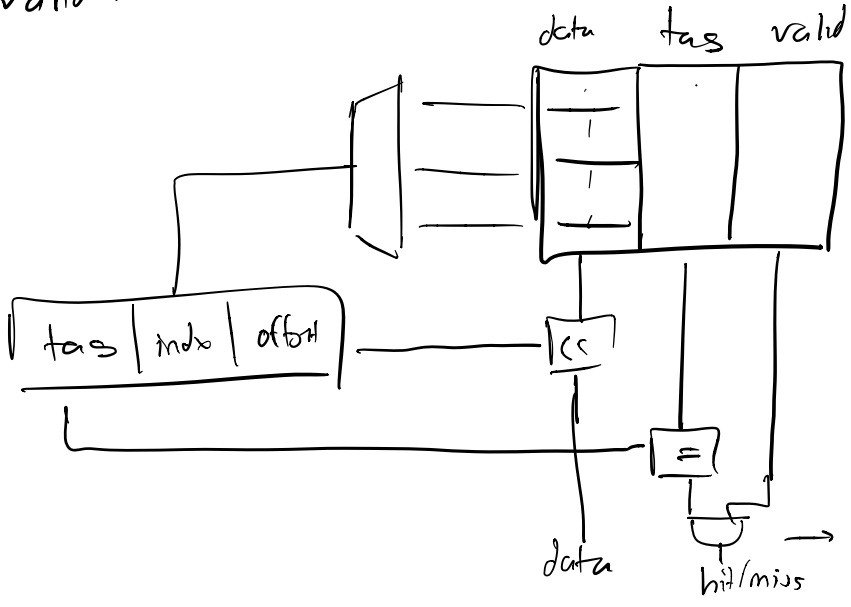
Offset bit: which byte in frame?
4B in frame ⇒ 2 bits.

Index: which set?
1024 frames ⇒ 1024 sets ⇒ 10 bits

② How to take data from cache?

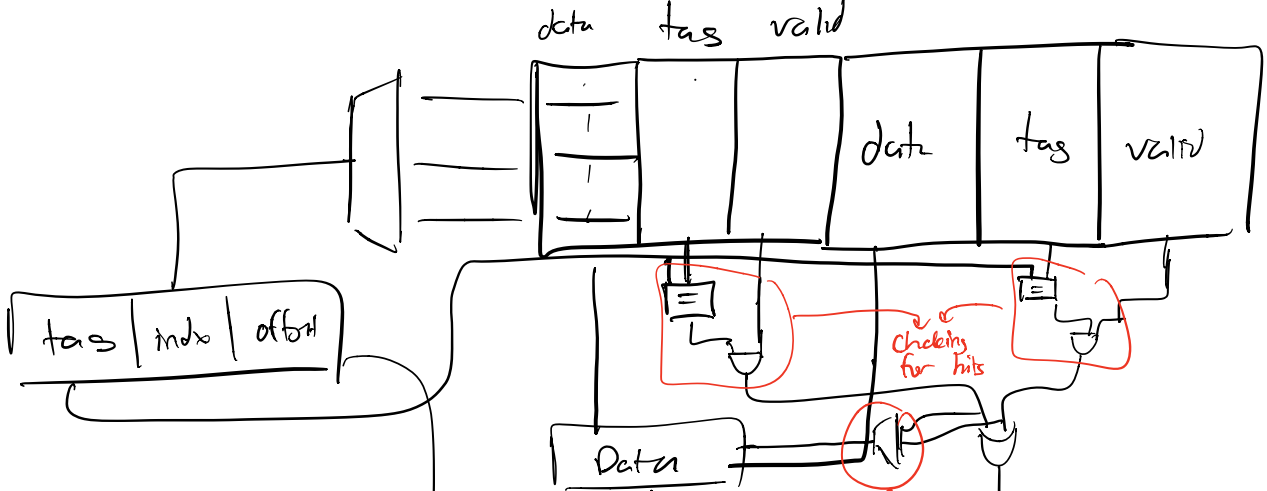


- Tag and valid?

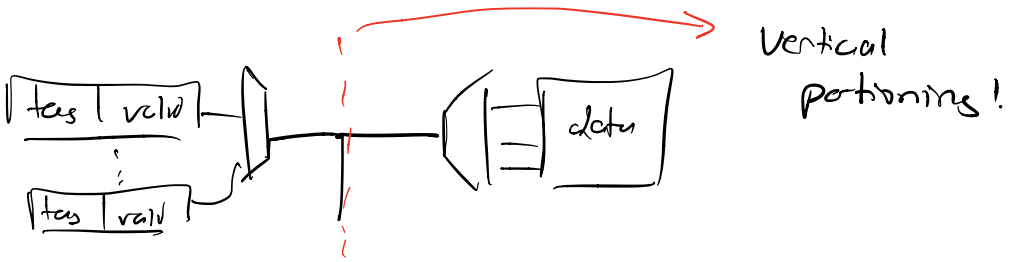


1 if both correct tag + valid?

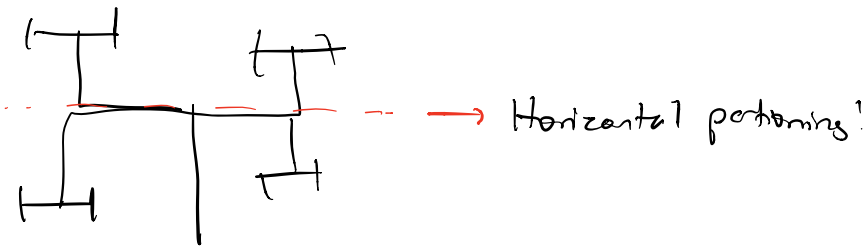
- Associative caches:



- Physical implementation (conserve material)

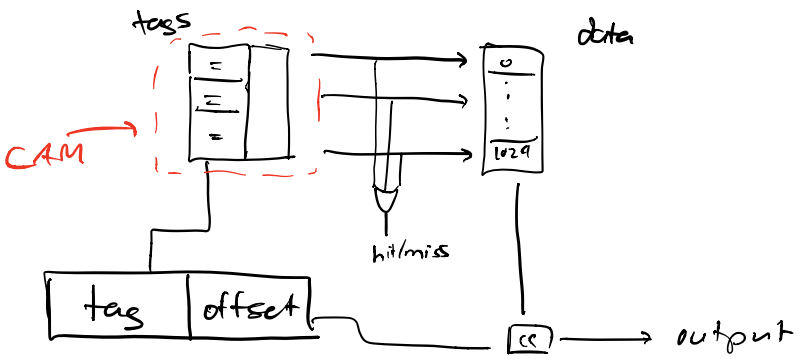


- High level:



- Fully associative cache:
  - Problem: prev. solution ⇒ too many gates + muxes.
  - CAM: content addressable memory. Expensive

Block → CAM → onehot encoding

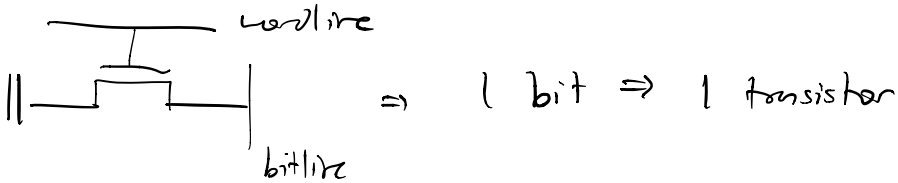


- Stores:
  1. Match tag
  2. Write to matching set

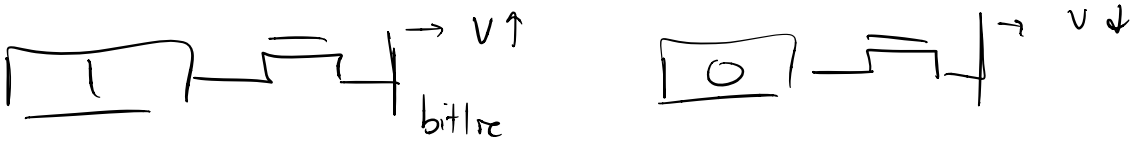
} Parallel → corruption of data

MAIN MEMORY IMPLEMENTATION

- Optimizing for density → DRAM. Same architecture



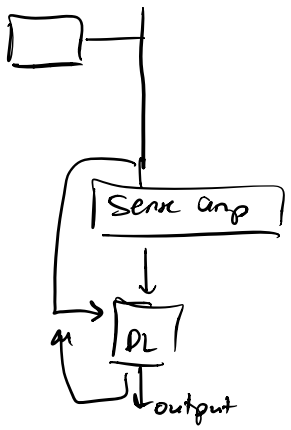
- Decay over time ⇒ DRAM must be refreshed
- Access time ∝ ports # × $\sqrt{\#\text{ of bits}}$ (Later than SRAM)

- Read:
  1. Precharge bitlines to $V_{dd}/2$
  2. Select word via wordlines
  3. 

  4. Use sense amplifier to detect difference

- Write:
  1. Select wordline + bit line
  2. Pass in data ⇒ charging/discharging capacitor

- Problem: destructive read (every time we read → data will change).
  - Soln: D latch as buffer.
  - Read:



1. Read
2. Write to buffer
3. Buffer will write back to cell (borrowing)

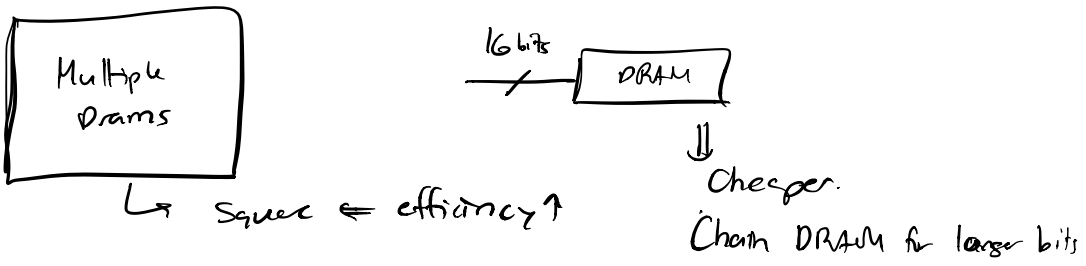
  - Write:
    1. Read word to buffer
    2. Write data into buffer
    3. Write buffer into cell

- Problem: leakage from capacitor
  - Soln: refresh DRAM

for all words → put into buffer → write back to DRAM

- Refresh interval ← capacity, tech., temp. (↑ temp → more refresh)
- 1-2% of DRAM time spent on refreshing

- Design choice for DRAM:



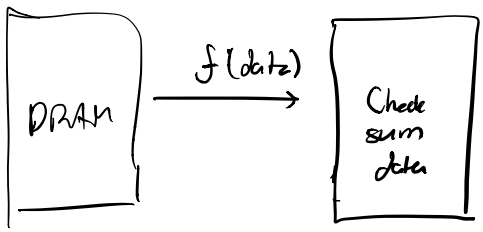
- Cycle time is 2× SRAM (refresh)
- Runs on diff. clock than processor ⇒ memory bus clock
  - If freq.↑ ⇒ memory bandwidth cannot handle

DRAM types

- Synchronized DRAM (SDRAM)
  - V. fast + efficient
  - DDR (double rate of DRAM)
- Graphics DRAM (GDRAM)
  - Larger bandwidth
- Rambus
  - Highly customizable (controller/port)
  - Expensive

- Dual inline memory module: DDR

- Problem: errors
  - Soln: checksum-style redundancy

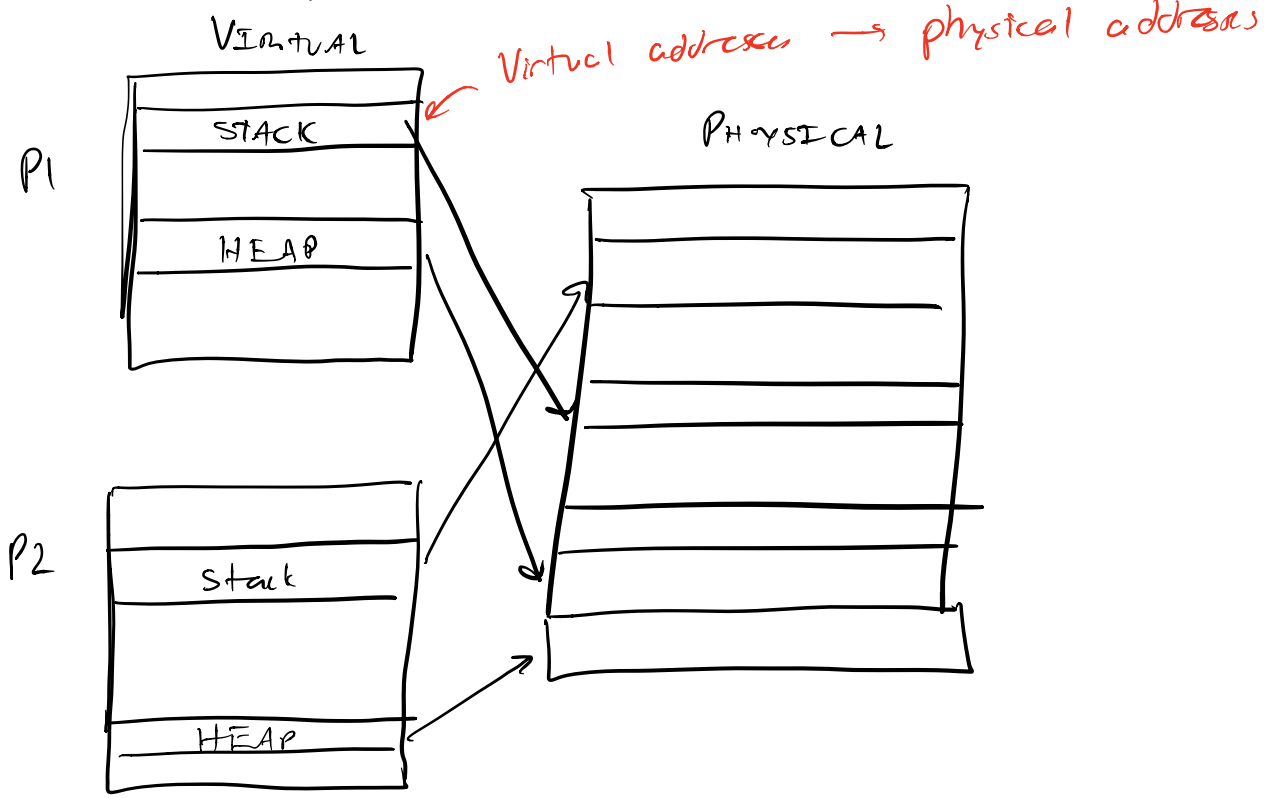


On R/W ⇒ check if f(data) = stored f(data)

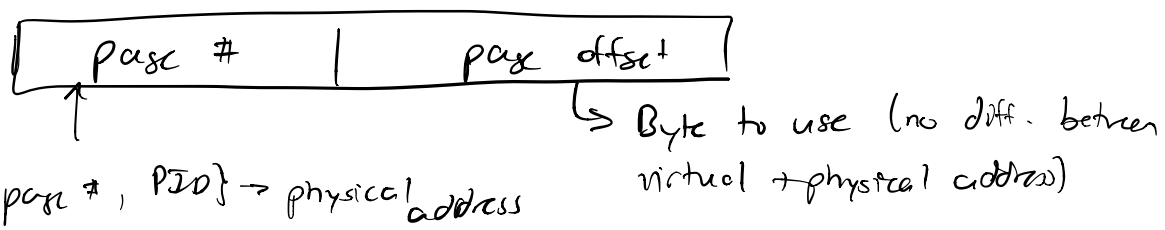
$f(\ldots)$: parity (XOR (bit). Work if 1 bit flip, not for even # of flips)

VIRTUAL MEMORY

- Memory is treated like cache ⇒ dynamic subset of total mem.
- Program thinks it has access to full memory ⇒ only has access to a couple of areas.



- Process:
  1. Assign virtual memory + get data
  2. OS + hardware: {process ID, virtual address} → physical address
     ↳ Virtual address can be same ⇒ need PID to distinguish
  3. Translation:

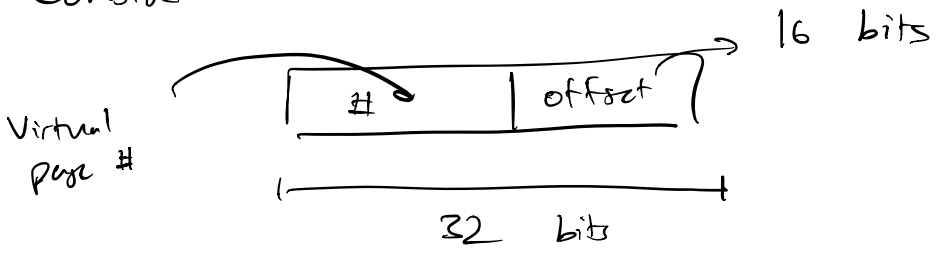


- Miss: page fault

1. Go to disk + get the data
2. Replaces page via LRU policy done by OS

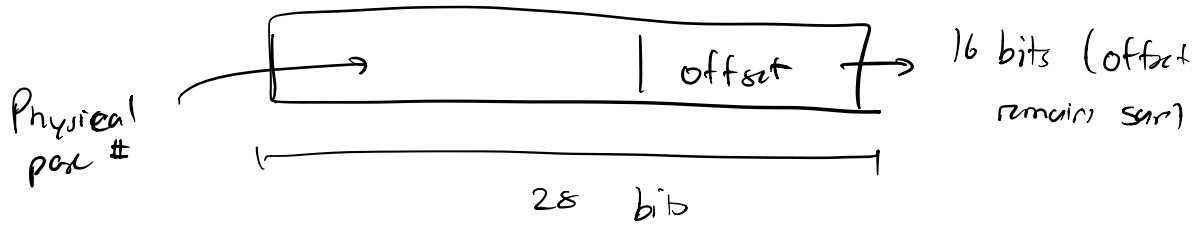
- Finding # of page # bits:
  - Ex:// 64 KB page → 2^{16} bytes → 16 bit offset for byte ID

Consider 32-bit arch:



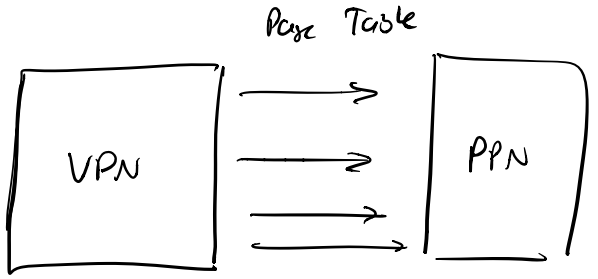
Consider 256 MB memory

To reference: 2^{28} bytes ⇒ 28 bits for physical addr



Page table: 16 bit of VPN → 12 bit of PPN

- How do we map between VPN + PPN?



⇒ Per process, stored in register

  - Ex:// 4B page table entries. 32-bit machine. 4KB pages. Page table size.

Q: How many entries in page table?

① Figure out entire memory capacity

32-bit VA → 4GB memory (virtual)

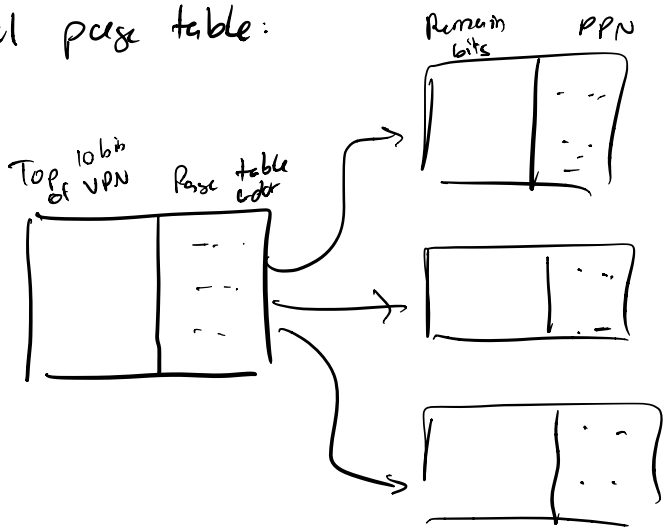
② How many pages?

$$\frac{4GB}{4kB} = 1M \text{ pages}$$

⑦ How many bytes per page entry?

$$1 \text{ mil pages} \times \frac{4B}{page} = 4MB \text{ page table}$$

- To reduce page table size:
  1. Increase page size: unused memory → more misses, longer time
  2. Multilevel page table:



B/c pages are big, not likely that we need all tables ⇒ saves space.

- Who does translation?
  1. Process translates own address
     - Page table would be in process virtual mem.
     - Problem: process owner has to manage access ⇒ messy
  2. OS translates:
     - Page table in OS virtual mem.
     - OS can determine access + when to use (L2 miss)

- L2 miss ⇒ interrupt
  - OS take control ⇒ VPN : PPN ⇒ return mem.

- Problem: takes time to get main mem. data
  - Soln: translation lookaside buffer (TLB)

Virtual addr.	Final mem. location

⇒ No need for page table

- Miss: use page table to get mem. location + update TLB

- How is page table implemented?

  1) OS: flexible organizatn, slow

  2) Hardware: not flexible, fast

- Ex:// 32 KB cache. 64 B block size. 4 KB page size. # of entries if we want similar miss rate in cache + TLB?

  Similar miss rate ⇒ covering same amount of memory.

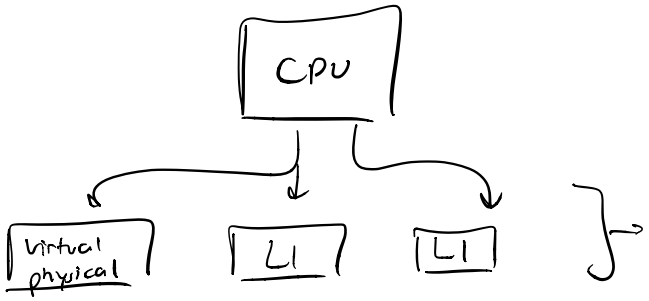
  Low-end: 32 KB ÷ 4KB page size ⇒ 8 pages

  High end: 32 KB ÷ 64 B block = 512 blocks (512 pages)

  ∴ TLB entries ∈ [8, 512]

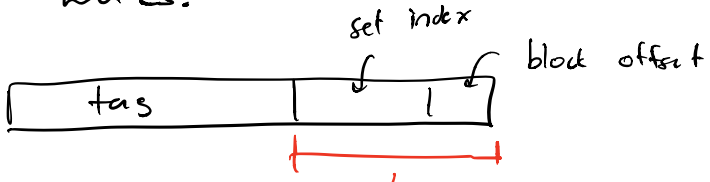
- Problem: B/c virtual addresses used to access cache ⇒ referring physical address/block in cache!

  - Soln: convert virtual addrss to physical at start (slow, but same pain)



2 cycles
1. Getting physical address from TLB + get set in cache
2. Compare physical tag of block

  - Reason why this works:

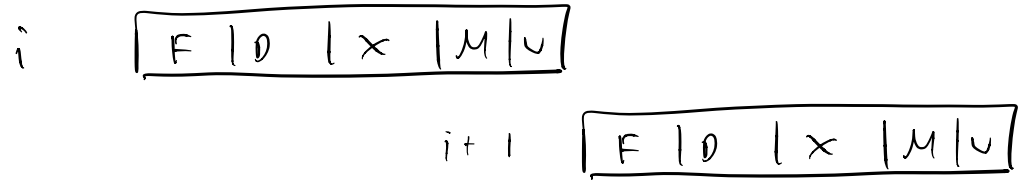


↳ If within page offset, this won't change even if physical/virtual

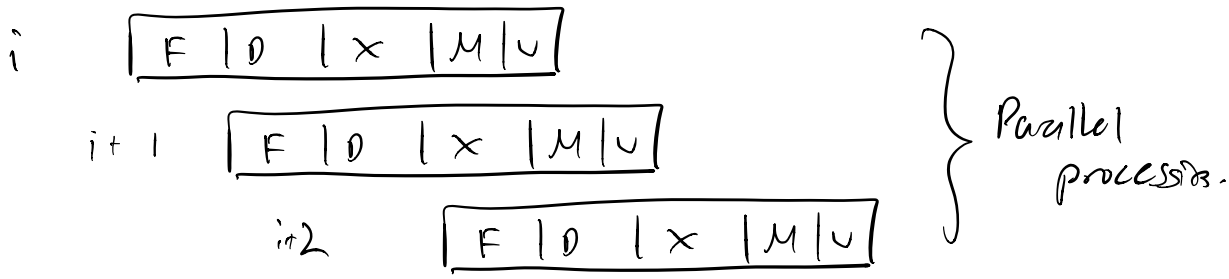
Constraint ⇒ more associative sets (less set bits)
smaller blocks (less block offset)
bigger pages (increase page offset)

PIPELINING

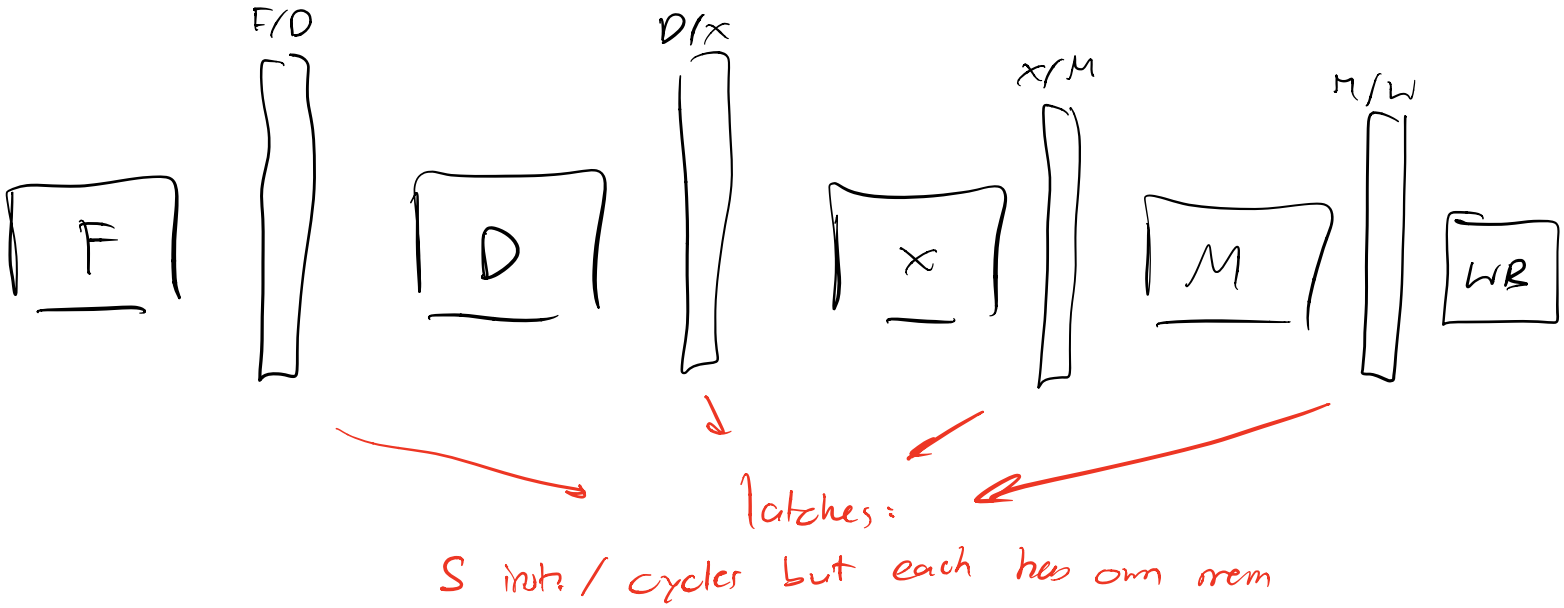
- Problem:
  1. single instr ⇒ single cycle ⇒ cycle period is long!
  2. Sequential processing slow
- Pipelining: throughput ↑
  - Non-pipelined:



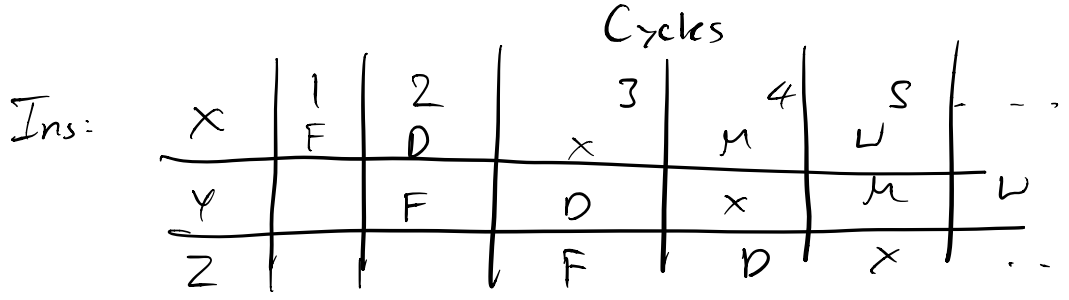
  - Pipelined:



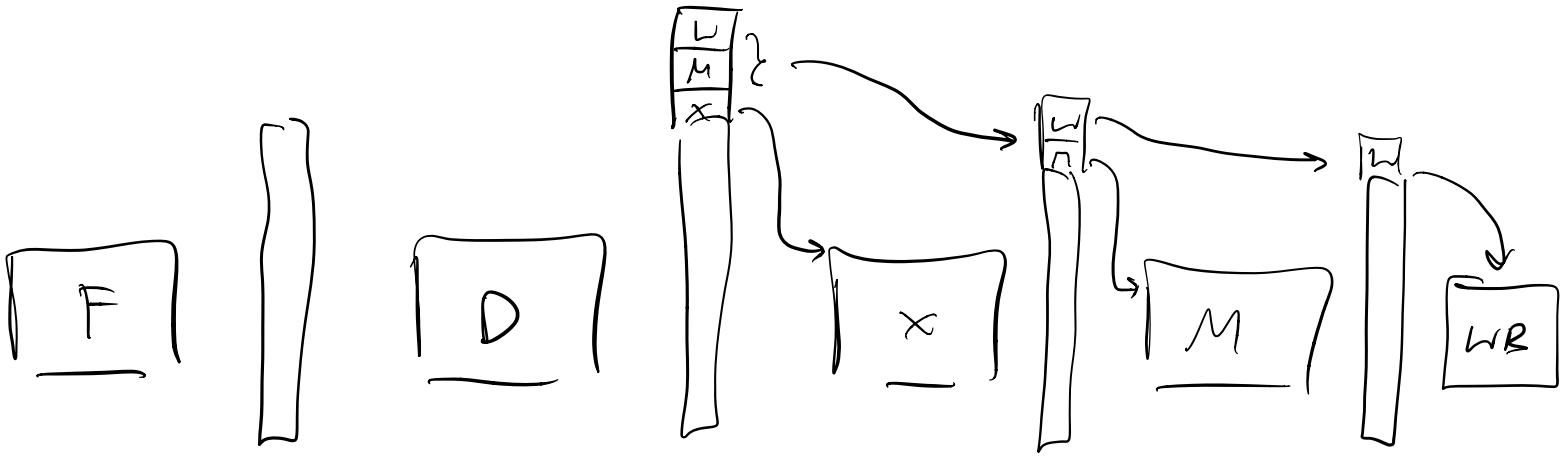
- Hardware:



- Pipeline diagram:



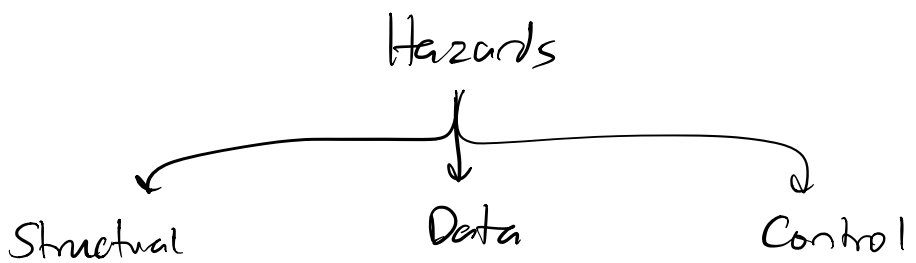
- Pipelining, controls for each instruction so that controls not mixed for different instructions.



- Performance of pipelined instructions:
  - Single cycle processor: performance = clock period
  - Pipelined processor: 1 clock/stage ⇒ clock period ↓

    Clock period > $\frac{\text{single cycle period}}{5}$ ∉ latches

- Hazards:



- Structural hazards: using same resource @ diff. stages

- Ex:// 1 instr. @ decode stage, another at write back stage

  Both using reg. file ⇒ use precedence

- Soln:

  1. Properly design ISA

     a) Each instr. uses a structure exactly once (ports, mux, hardware component)

        - In ex, have diff ports in reg. file

     b) Each structure uses struct exactly once

     c) Instr. in only 1 stage

- Data hazard

  - Dependencies: instr. depends on data from another instr.

Read after write

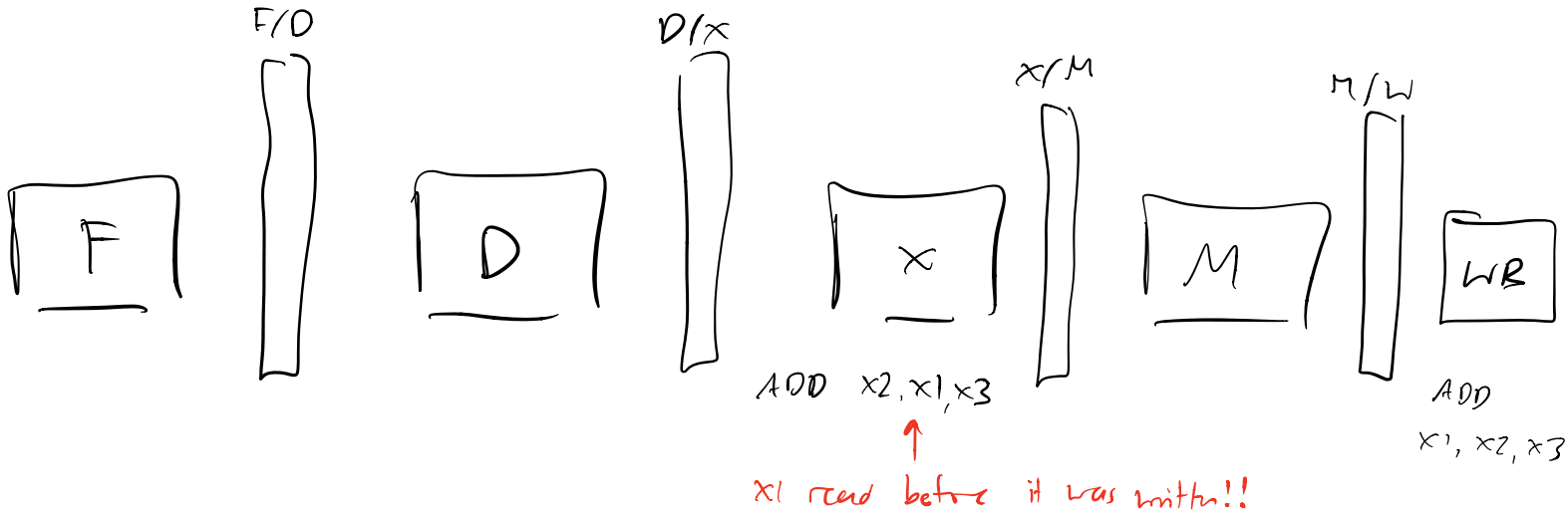
→ ADD x1, x2, x3
LDR x4, [x1, #4]

Write after write

ADD x1, x2, x3
ADD x2, x1, x3

Write after read

LDR x4 [x1, #4]
ADD x1, x2, x3



- If problem where: no solutions to above problem + they give sequence of instructions. Will it work?

  ① Find data dependencies in sequence.

  ② Use pipeline table as you move cycle ⇒ update registers.

- Tip: if you read < 3 cycles after write instr. ⇒ data hazard

- Soln: stalls s.t. get correct answers / bypassing

Stalls

Software interlocks

- Compiler puts indep. instructions / NOPs to prevent hazard
- Ex:

```
ADD x3, x2, x1   ①
LDR x4, [x3]     ②
STR x7, [x3]     ⑦
ADD x6, x2, x8   ④
```

①	F	D	X	M	W	
④		F	D	X	M	W
②			F	D	X	M W
③						

This still leaves us 2 hazard, so add NOP ⇒ delays for 1 cycle

- Performance: not great

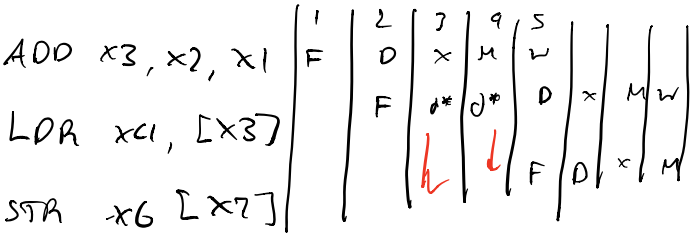
  20% instrs need NOP, 5% need 2 NOPs

$$CPI = 1 + \underbrace{0.2 \times 1}_{1\ NOP} + \underbrace{2 \times 0.05}_{2\ NOP}$$

Hardware interlocks

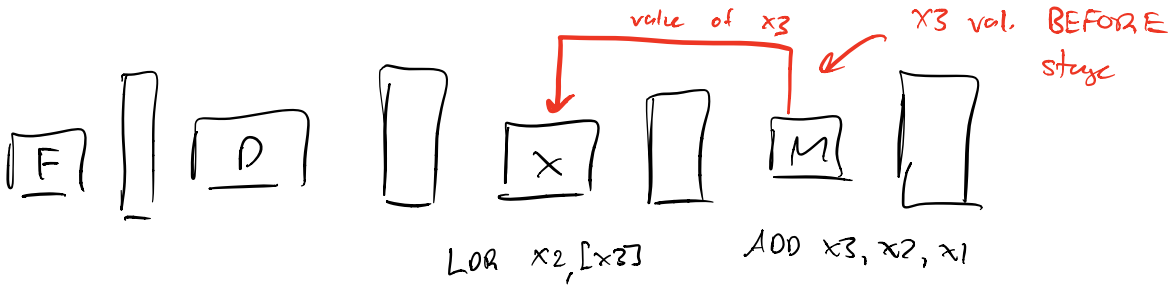
- More compatible ⇒ indep. of # of stages
- Detect hazards:
  1. Pass instruction reg. w/ latch
  2. Compare names w/ comb. circuit (stall logic)
  3. Delay if same name is used!
- Fix hazards:
  1. add NOP into D/X latch
  2. Clear controls
  3. Disable F/D latch + PC write (no future instr. should be affected).
- Ex:

```
ADD x3, x2, x1
LDR x4, [x3]
STR x6 [x7]
```

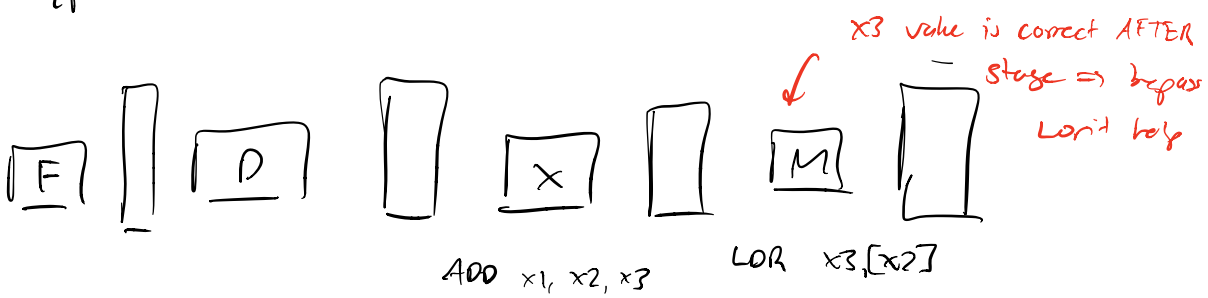


Delay propagates!

- Bypassing: reduces stall delay by linking stages



  - Add bypass logic: select between normal input / bypass input
  - Bypass first, then stall if cannot bypass



    - If value after latch is NOT correct for bypass → stall

- Control hazards: need to get next instr. w/out knowing outcome of branch

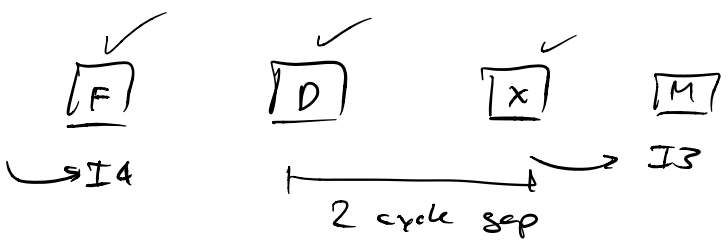
  - Default solution:

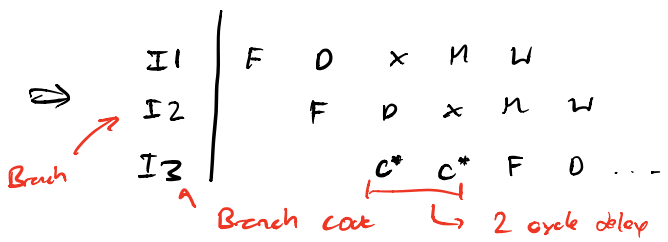
Assume branch not taken & fetch next line of code

Branch not taken ✓

Branch is taken (ALU)

2 cycle delay → Flush previous latches + fetch branch code ← Branch recovery





  - Performance: since so many taken branches → CPI ↑
  - Optimizations:

1) Fast branch: move up branch calc. in decode stage

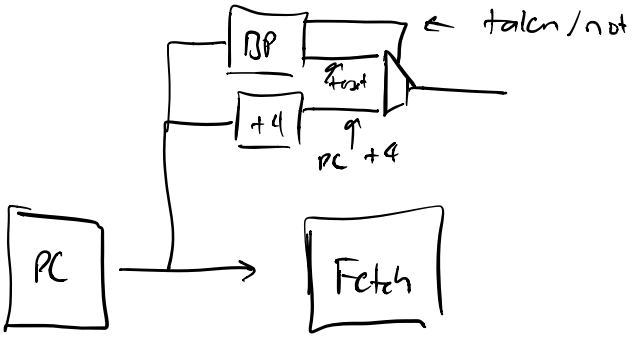
- Pros: flush out only 1 instruction ⇒ delay ↓
- Cons: Need to support bypassing into decode
- Performance: CPI $1 + (\%\text{ of branches})(\%\text{ taken})(1\text{ cycle})$

2) Delayed branch: compiler puts indep. insns after branch ⇒ no flush

- Pros: delay ↓
- Cons: ISA modification + compiler
- Performance (CPI): $1 + (\%\text{ branch})(\%\text{ taken})(1\text{ cycle}) +$

$(\%\text{ branch})(\%\text{ delay or NOPs})(1\text{ cycle})$

3) Dynamic branch prediction: predict next instruction

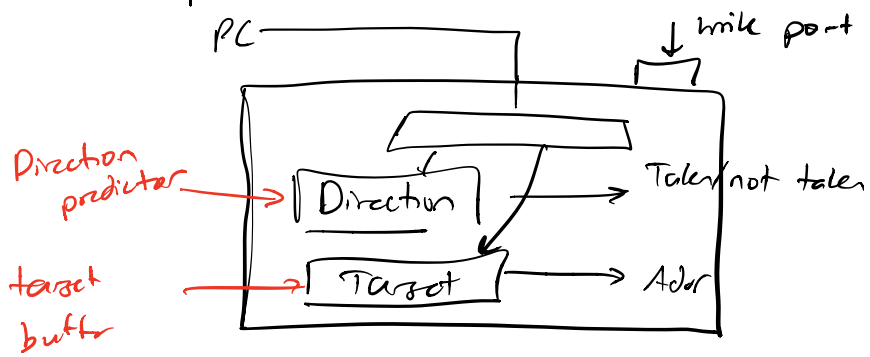


If wrong:

2 cycle flush

(Only know if in the execute stage ⇒ F+D stage must be flushed)

Branch predictor:

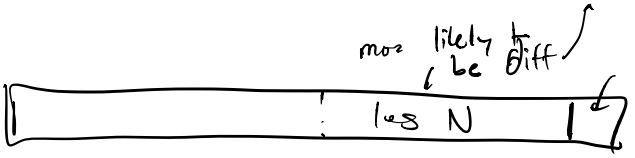


1. What part of PC are we using for prediction?

a) Number of data entries in buffer: N

∴ log N bits to index

b)



2 bits = 0

1/2 PC is in terms of words

2. What happens if 2 PCs have same target/direction b/c $\log N$ bits same?

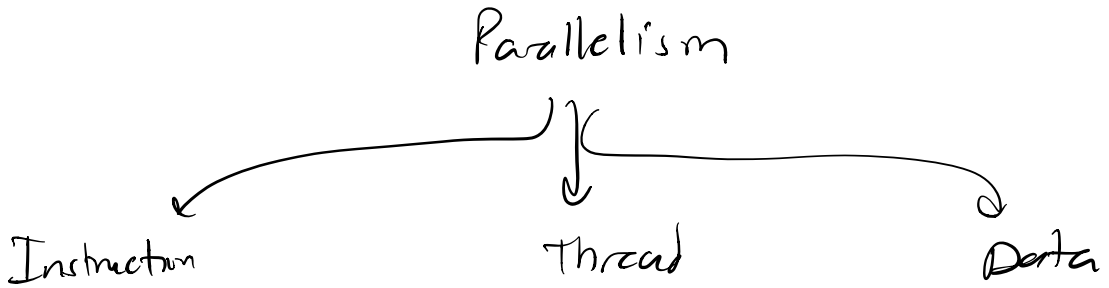
↳ Aliasing ⇒ not a problem b/c just prediction! :)

3. Updating predictions: wait port for incorrect pred.

Performance wise:

CPI: 1 + (% branch)(% mispredicted (v. low)(2 cycle)

- If # of stages in pipeline ↑:
  - Clock cycle ↓, but delays longer (more stages to correct)



- Pipelining
- Superscalar (multiple instr. / stage)
- Out of order execution