$$P\{X \in A\} = P\{X \in A, Y \in (-\infty, \infty)\}$$

$$= \int_A \int_{-\infty}^{+\infty} f(x,y)\, dy\, dx$$

$$= \int_A f_x(x)\, dx$$

```
CREATE USER 'bhadresh'@'localhost' IDENTIFIED
                                   BY 'bhadresh'.
GRANT ALL PRIVILEGES ON '<name>'.*
                     TO 'bhadresh'@'localhost'

FLUSH PRIVILEGES
```

THE PROCESSOR

* brain
* compute, decide, control flow

performance :

* inst. count (IC) – n(inst. exec. for prgrm)
  ISA ⟶ IC, richer ISA ⟶ ↓IC, ↑inst. complexity
  compilers ↓IC by optimizing code
* CPI (cycles / inst) – mean(clk. cycles / inst)
  inst. types ⟶ n(cycles) {arithmet, memory acc., branch, etc)
  hardware design ⟶ CPI

* cycle Time (CT) – duration (each clk. cycle)
  processor's clk. freq ⟶ CT (3 GHz ≡ 3×10^9 inst/sec)
  longest delay path ⟶ CT
  (seq. of components that takes the longest time for signals to propagate through – slowest route.
  for the processor to fn., it must accommodate this as well, hence longest path delay ⟶ max clk. freq, hence CT as well)

CPU Time ⟶ time taken to exec all inst

$= n(\text{inst}) \times t_{\text{per inst}} = n(\text{inst}) \times \text{cycles}_{\text{per inst}} \times \text{time}_{\text{per cycle}}$

$= IC \times CPI \times CT$

∴, **CPU Time = IC × CPI × CT**

MIPS Implementation approaches:

* two implementations
  - single-cycle (1 inst ≡ 1 cycle)
  - pipelined (multiple inst. processed simult. in diff stages)

* MIPS inst. subset focus: ↳ lw (load word), sw (store word)
  ↳ add, sub, and, or, slt (set less than)
  ↳ beq (branch if eq.), j (jump)

R-format –
a) reg-reg
b) form: [opcode] [rs] [rt] [rd] [sh. amt] [funct]

add, sub, lw, ... ↑ source ↑ ↑ dest ↑ shift amt ↑ operation/code of fn.

Instruction Execution Process

* Fetch: Use prog. counter (PC) to get inst. from mem.
* Decode: Det. type of inst. & read registers
  (det. type → using some binary pattern, opcode - based on this, diff. control signals will configure datapath, activated by processor
  read register → 32 reg.s - 5-bits req.d to specify which to use - this number is sent to reg. file during decode - file o/p the values stored in those reg.s, & vals are sent to ALU for operations)
* Execute - ALU → arithmetic/logic, calc. mem. addr, calc. branch target.
* Memory Access - for lw/sw inst.
* Update PC - move to next inst. at (pc+4) / branch target.

Load/Store instruction

example -

take lw $t0, 12($s1) → load whatever's at addr. ($s1 + 12) into $t0 reg.

1) get base addr (say $s1 = 1000 is addr.)
2) calc. eff addr. (1000 + 12 = 1012 → val at this addr. to be fetched).
3) acc. memory (go to addr = 1012, & read the 32 bit val there, say 42)
4) put value into $t0 (i.e., $t0 = 42)

load:

inst. from I.M. → extract reg. no's & offset (source, dest, offset val) → read val. of source reg from Reg. file (only $s1)

↓

extend offset to 32-b, from 16-b

↓

ALU does offset addn.

↓

address sent to Data mem

↓

Data mem returns val at addr. → value WB to reg.

Store: sw $t0, 12($s1)

inst. from I.M. → extract reg. no's & offset (source, dest, offset val) → read val. of source reg's from Regfile (both $t0 & $s1)

↓

extend offset to 32-b from 16-b

↓

ALU does offset addn. ($s1 + 12)

↓

address sent to data mem (Addr. i/p)

↓

$t0 is directly fed from reg. file to mem to data i/p (memWrite = 1) → STORED!!!

inst ≡ 32-b

6 bits	5 bit	5 bit	16 bit
opcode	src r_t	target r_d	offset

± values → can cover almost everything

Branch Instruction

* focus - beq.
* beq $t0, $t1, offset

1) IF - fetch from I-M at addr. in PC
   - PC+4 done for next inst.

2) ID - opcode = 4 for branch
   - reg no.s gotten ($t0, $t1, basically addrs)
   - 16 bit offset gotten

3) CMP - $t0, $t1 values gotten from reg. file
   - ALU subtracts them.
   - flag = zero, if res = 0. (equal)

4) Branch - 16 bit offset extended to 32 bits
   - shifted left by 2 bits, as:
     * inst ≡ words, word ≡ 4 bytes
     * offset → unit of words, not bytes.
   - this is added to PC+4 to form target addr.
     (offset → relative to inst. following the branch. So, its PC+4 + sign_extended offset<<2)

5) Branch Decision - if ALU ≡ Zero (set) then branch target
   - else, PC = PC+4 remains
   ↳ MUX is fed with ALU Zero val.

Composing Datapath Elements

? Why compose — single design supports all inst.
(R-format, Load/store, Branch)
— to consider:
* Shared resources
* mux
* control signals

Key design decisions -

1. Separate Inst. & Data Memories:
* can't use same mem. for both inst in single-cycle design
* IM during IF, DM during EX/MEM (no conflicts here) (harvard architecture)

2. Mux placement:
* ALU input - select b/w 2nd reg. value (for R-type) & sign-extended offset (load/store) {lw/sw $t0, x($s)}
* Reg write (Data) - select b/w ALU o/p (R-type) & mem data (load)
* Reg write (Addr) - select b/w r_d field (R-type) & r_t field (load)
consider: add $t0, $s1, $s2
lw $t0, 12($s1).

design problem. ← but
1 write i/p, 2 diff. fields.
↳ dep. on type of inst.

- in add - $t0 is specified in r_d field
- in lw - $t0 is specified in r_t field.

Reg. Write Addr → 2i/p : r_d field in inst (bits 15-11)
r_t field in inst (bits 20-16)

→ control signal : 0 → r_d (R-type)
1 → r_t (lw)
if no write → 0

add r_d, r_s, r_t, sh→5, fn→6 | R-type
lw r_t, off(r_s) | load

→ select WHERE to write, based on control signal (RegDst=1 → rd, else r_t)

* PC-input : selects b/w PC+4, branch target & jump target ← direct location

3. control signal Distribution :
   - ↳ RegWrite
   - ↳ ALU Op
   - ↳ MemRead
   - ↳ MemWrite
   - ↳ Reg-ALU MUX
   - ↳ D-Mem-Reg MUX
   - ↳ Branch mux

? jump not included.

COMPOSED DATAPATH

IF
* PC → inst. addr.
* inst fetched from I-M
* PC = PC+4

→

ID
inst. decoded into fields
register operands fetched

→

EX
* ALU → operation
* R-type: ari/logic
* lw/sw: addr. calc
* branch: CMP

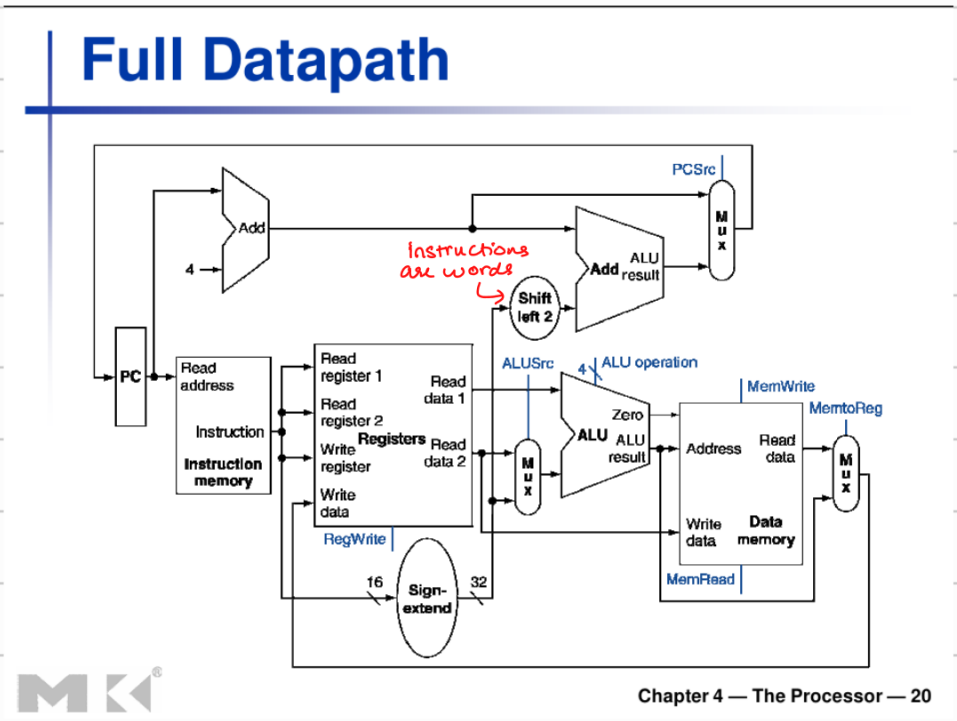
↓

MEM (if reqd)
* lw : read mem
* sw : write mem
* other : bypass mem

←

WB (if needed)
* R-type & load : WB to Reg file
* branch & sw : NOWB

இதுவரை நாம் கற்றுக்கொண்டது (whatever we've done till now):



SIMPLE IMPLEMENTATION SCHEME

* control logic: to know what the inst. is & what each component shd. do for that inst.

* 2 control units ⟶ Main, ALU.
  - ↳ Main: * takes opcode (bits 31-26)
    * generate control signals for regWrite, memRead/Write, ALUOp, RegDst, MemtoReg, ALUSrc, PCSrc & jump/branch
  - ↳ ALU: * needs to know what op to perform (add, sub, ..)
    * this control is not directly from opcode, as R-type ≡ SAME OPCODE!
    * ∴, ALU control gets ALUop & funct field from inst

↑ MODULARITY

funct → decides op.

funct ↓ (5-0 bits for R-type)

opcode	ALUOp	Operation	funct	ALU function	ALU control
lw	00	load word	XXXXXX	add	0010
sw	00	store word	XXXXXX	add	0010
beq	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than	0111

R-type inst : (e.g. add $t0, $t1, $t2)

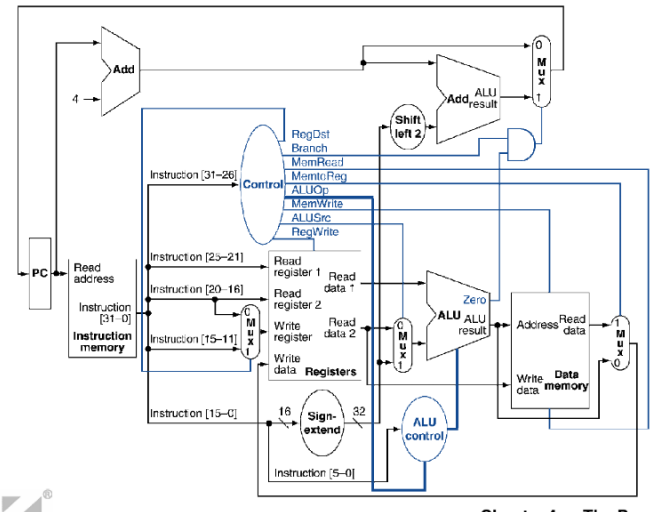
* opcode = 000000
* RegDst = 1 (dest is r_d)
* ALUSrc = 0 (use reg. value, not offset)
* MemtoReg = 0 (no write from memtoReg)
* RegWrite = 1 (writing to rd, so ye)
* memRead | Write = 0
* ALUOp = 10 (let ALU decide based on funct)
* ALU control gets funct.
  funct = 100000 → ALU control o/p = 0010
  = 100010 → sub → control o/p = 0110
  = 100100 → and → 0000
  & so on ...
* result = 2 vals added & stored at r_d.

Load inst : (lw r_t, off(r_s))

* opcode = 100011
* main control:
  - ↳ memtoReg = 1
  - ↳ readMem = 1
  - ↳ writeMem = 0
  - ↳ ALUSrc = 1 → use offset.
  - ↳ ALUOp = 00 → basic addr. addn.
  - ↳ RegWrite = 1
  - ↳ RegDst = 0 (write to r_t)
* ALU control: 00 → just add, so base + offset done & fetch word from memory, writes into r_t.

Branch-On-Equal : * opcode = 001000
(beq $r1, $r2, label) * main control :
↳ WriteReg = 0
↳ Write/ReadMem = 0
↳ ALUSrc = 0
↳ ALUOp = 01 → subtract
↳ Branch = 1
* ALU : ALUOp = 01 → subt.
if $r_1 - r_2 == 0$, then PC → branch target.

Datapath With Control

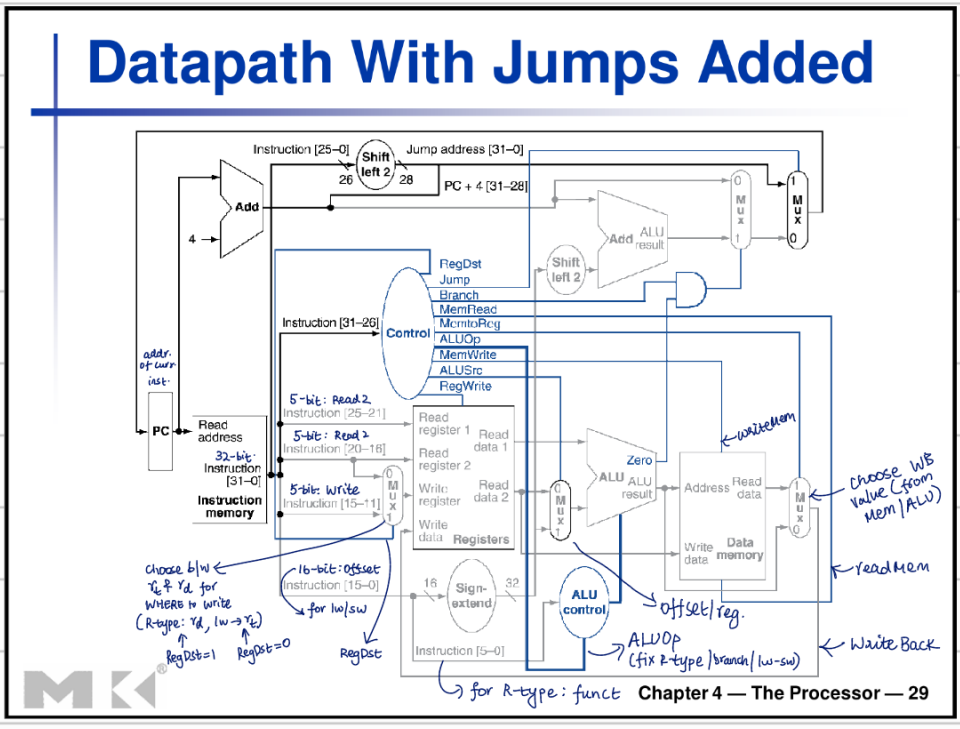


Signal	Purpose
RegDst	Selects destination register: rd (R-type) or rt (load)
RegWrite	Enable writing to register file
ALUSrc	Use immediate or register as second ALU input
MemRead	Enable reading from memory (load)
MemWrite	Enable writing to memory (store)
MemtoReg	Selects value to write to register: from memory or from ALU
Branch	True if instruction is a beq
Jump	True if instruction is a j → shall address
ALUOp[1:0]	Sent to the ALU control unit

MK

? How are jumps handled
* take upper bits of PC+4
* concatenate them with 26-bit addr. left shifted by 2. (again, inst. are words, so ye)
* PC needs a 3-way MUX for:
↳ PC+4
↳ jump addr.
↳ branch addr.
* new control signal for jump.

} addrs. to the structure.



PERFORMANCE ISSUES :

* slowest instruction $\xrightarrow{\text{determines}}$ clock cycle time

(e.g.) lw : I·M → Reg·Read → ALU → MEM → Reg·Write

↳ LONGEST DELAY path!

∴, simple insts. like add shd. also wait.

Hence, INEFFICIENT & VIOLATES RISC principle → Make the common case fast.

PIPELINING

* overlapping of multiple inst.

analogy : say ∃ 1 washer, 1 dryer, 1 folder. instead of waiting one full load to go thru all before starting next, you overlap :

Time	Washer	Dryer	Folder
0-30	Load 1		
30-60	Load 2	Load 1	
60-90	Load 3	Load 2	Load 1
90-120	Load 4	Load 3	Load 2

You finish **4 loads in the time it would've taken to do 1.**
→ That's pipelining!

w/o pipelining ⟶ CPU waits ∀ 5 stages before next one
⟶ 1 inst ≡ 5 cycles
⟶ other units idle while not being used
⟶ waste of hardware.

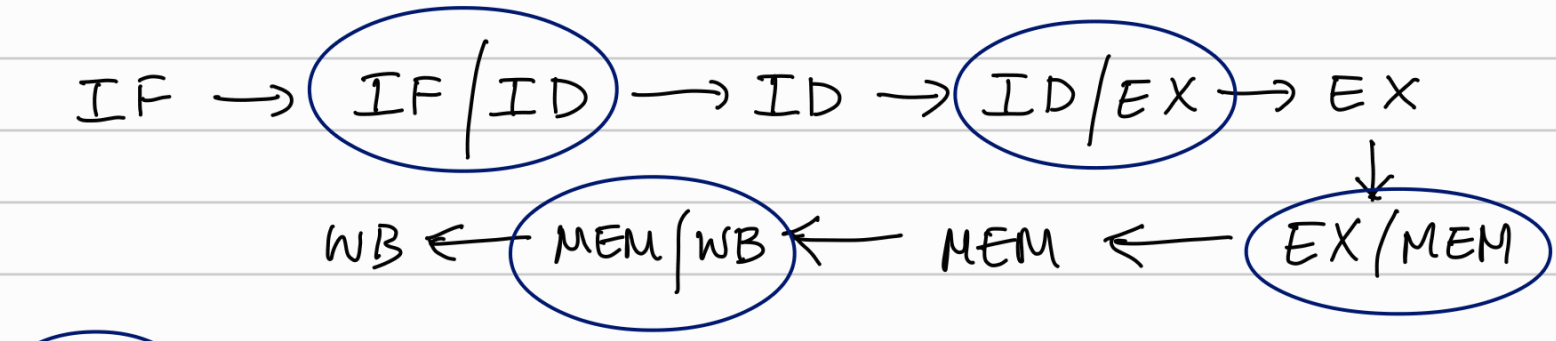
with pipeline ⟶ each stage of datapath is ALWAYS DOING USEFUL WORK.
⟶ means, 1 inst ≡ 1 cycle, after initial ramp up!!!

Applying to MIPS :

Cycle	IF	ID	EX	MEM	WB
1	Instr 1				
2	Instr 2	Instr 1			
3	Instr 3	Instr 2	Instr 1		
4	Instr 4	Instr 3	Instr 2	Instr 1	
5	Instr 5	Instr 4	Instr 3	Instr 2	Instr 1

Stages ⟶ IF, ID, EX, MEM, WB
imagine pipelining this ⟶ each inst ≡ 5 cycles, but new inst starts EVERY cycle
this works because each inst. uses different part of the datapath at each stage. So, we can:
↳ Split datapath with pipeline registers b/w stages.
↳ feed new insts. into pipeline as prev. ones move forward.

to keep insts. from overwriting each other, we insert pipeline registers. essentially these are chk. pts.



⬭ → pipeline registers.
* hold o/p of prev stage
* feeds i/p to next stage in the NEXT clk cycle
* keeps diff. insts. data separate.

∴, EACH INST moves forward one stage/clock, like water flowing step by step down a stair stepped waterfall

Performance Intuition

* throughput - {n(comp. inst/unit time) → speed of pipeline.

w/o pipeline → 1 inst ≡ 5 cycles
∴, throughput = 0.2 inst/cycle

with pipeline → 1 inst ≡ 1 cycle (after ramp-up)
∴, throughput = 1 inst/cycle
⇒ ~ 5 x the speed.

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

Actual Numbers (Slide 33)

Assume:
- IF, EX, MEM = 200 ps
- ID, WB = 100 ps

Without pipelining:
- `lw` takes 800 ps (200+100+200+200+100)

With pipelining:
- Each stage is **split into a 200 ps slice**
- Instructions flow in at 200 ps intervals

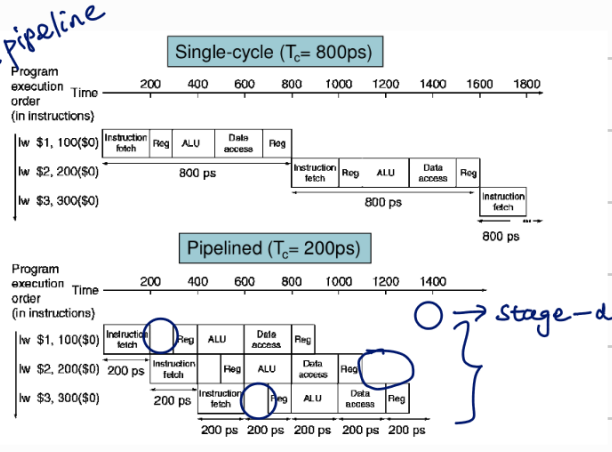
Now you complete **one instruction every 200 ps**
→ **4× speedup over 800 ps single-cycle time**

(Why not 5×? Because of **pipeline fill time** and **unbalanced stage delays**.)

time b/w inst. in pipeline

800/200

↑ AFTER ramp-up time (1st inst fin.)



HAZARDS

* stall | disrupt the flow

STRUCTURAL HAZARDS : * two stages accessing same hardware (e.g. IM & DM)
* Sol: use diff. mems for both (IM, DM)

DATA HAZARDS : * an inst. depends on the result of prev one.

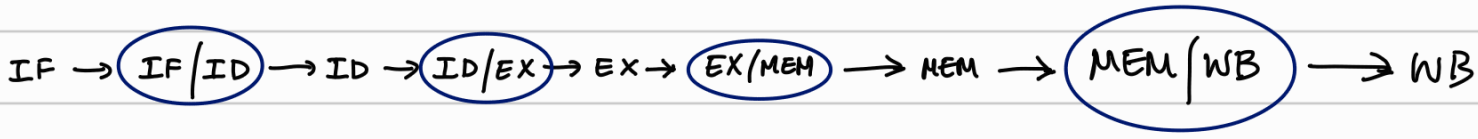
(e.g.) add $t_1, $t_2, $t_3
sub $t_4, $t_1, $t_5 ← needs $t_1 before it's ready.

CONTROL HAZARDS : * next inst. depends on result of branch
* won't know branch status until EX or MEM.

DESIGN CONSIDERATIONS

↳ all inst: 32 bits ⟶ eZ to decode
↳ few inst formats → uniformity
↳ reg. operands in fixed positions → faster decode
↳ load/store - only memory access - memory stage is predictable

PIPELINED DATAPATH & CONTROL



What Each Pipeline Register Stores

Let's look at what data needs to be passed from one stage to the next:

➤ `IF/ID` **(after Instruction Fetch):**

- Instruction (to decode) | w/o this, we won't know what was decoded
- PC+4 (used for branch calculation later)

➤ `ID/EX` **(after Decode):**

- Register read values: `ReadData1`, `ReadData2`
- Immediate value (sign-extended)
- Destination register numbers (`rt`, `rd`)
- Control signals: ALUOp, ALUSrc, etc.

in EX stage:
* do ALU
* where res. to go
* write to Mem/Reg

w/o this, dk which operands to compute / (what to do)

➤ `EX/MEM` **(after Execute):**

- ALU result
- Register value for `sw`
- Destination register number
- Control signals: MemRead, MemWrite, MemtoReg, RegWrite

in MEM, we need:
* addr & data for sw
* addr to read for lw
* almost nil for R-type

w/o diz, mem. be like: whose res. is this? Smd. I even write to Mem.

➤ `MEM/WB` **(after Memory access):**

- Data read from memory
- ALU result (for instructions that don't access memory)
- Destination register number
- Control signals: MemtoReg, RegWrite

decide what & where to write

Control Signals are stored to tell the datapath how to handle the instruction at that stage

each inst → in diff. stage. So,
* Keep its OWN DATA (regval/addr)
* Carry control signals (know what to do)
* not mess with data from inst before/after it.

pipeline regs → backpacks

pipeline regs are like a baton in a relay race.

- Once an instruction finishes a stage (say, ID), it **hands off its baton** (data + control signals) into `ID/EX`.
- That baton is **only used by the EX stage** during the **next clock cycle**.
- After that, it's **no longer needed by EX**, and gets **replaced** with the next instruction's baton.

✳ So you're not relying on **reuse** of old data—you're ensuring that **every instruction has a fresh and private copy** of what it needs, just in time.

Assurance Comes from the Clocking Discipline

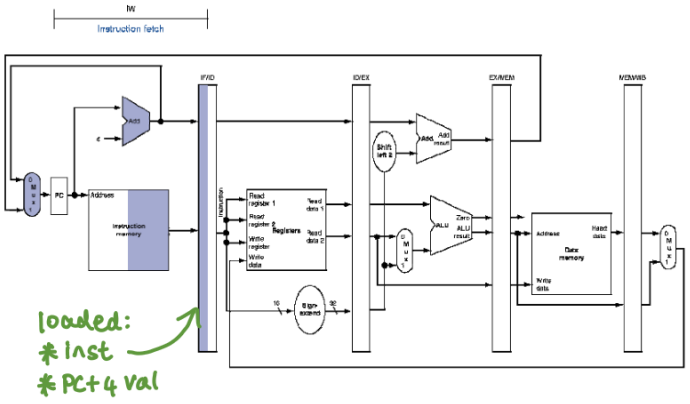
The datapath is designed using a **strict synchronous clocking model**:

- At every rising edge of the clock:
  - Each stage **writes its output** into the next stage's pipeline register.
  - Each stage **reads its input** from its own pipeline register.

This provides a kind of **"lockstep" movement**, where:

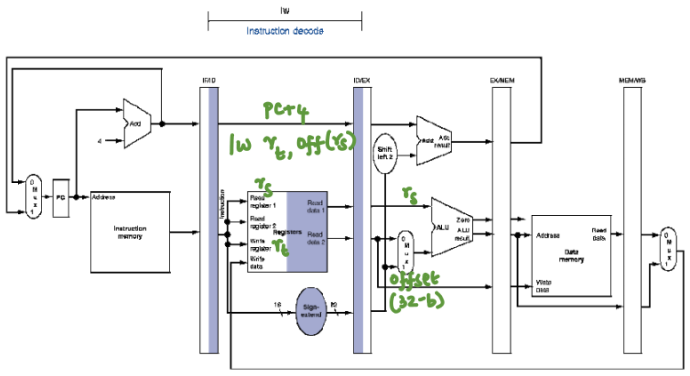
- **Stage N writes → Stage N+1 reads → Register overwritten**
- But the read happens **before** the overwrite—**guaranteed** by the clock cycle boundaries.

IF for Load, Store, …



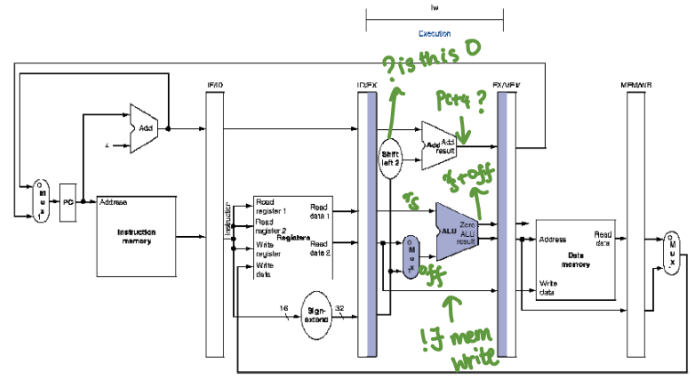
loaded:
* inst
* PC+4 val

ID for Load, Store, …



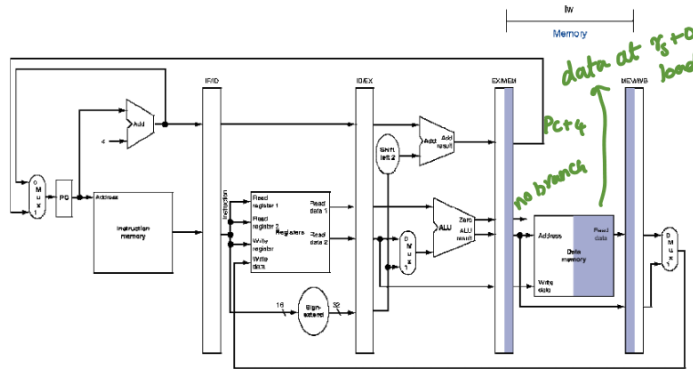
PC+4
lw r_t, off(r_s)
r_s
r_t
offset (32-b)

EX for Load



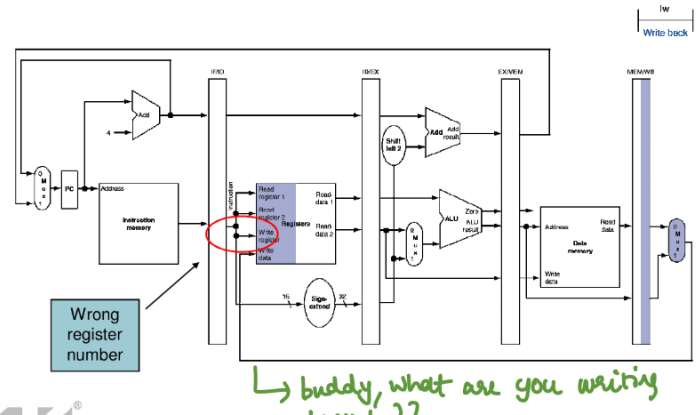
is this D
PC+4 ?
r_s+off
r_s
off
if mem write

MEM for Load



data at r_s+off loaded
PC+4
no branch

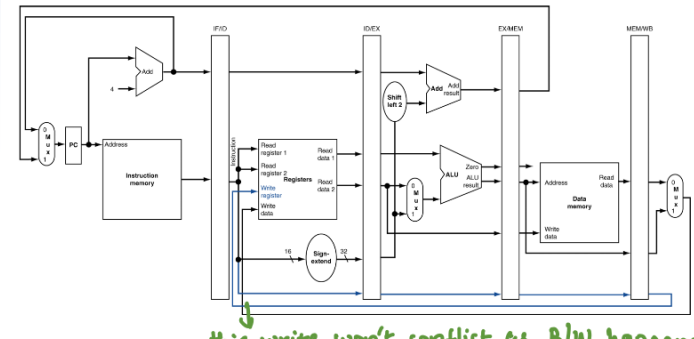
WB for Load



Wrong register number

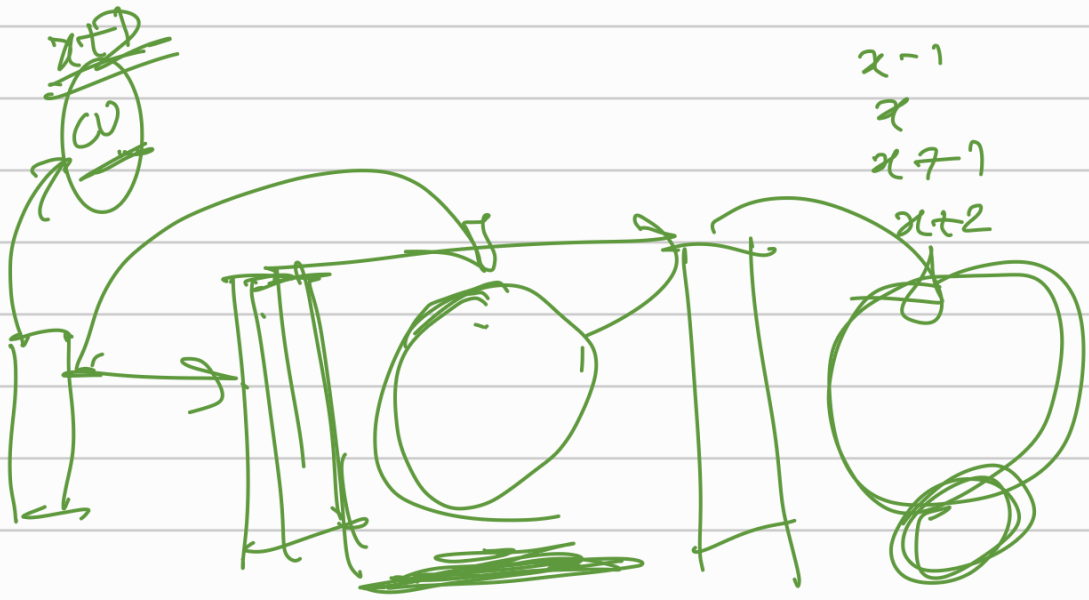
↳ buddy, what are you writing back to??

Corrected Datapath for Load

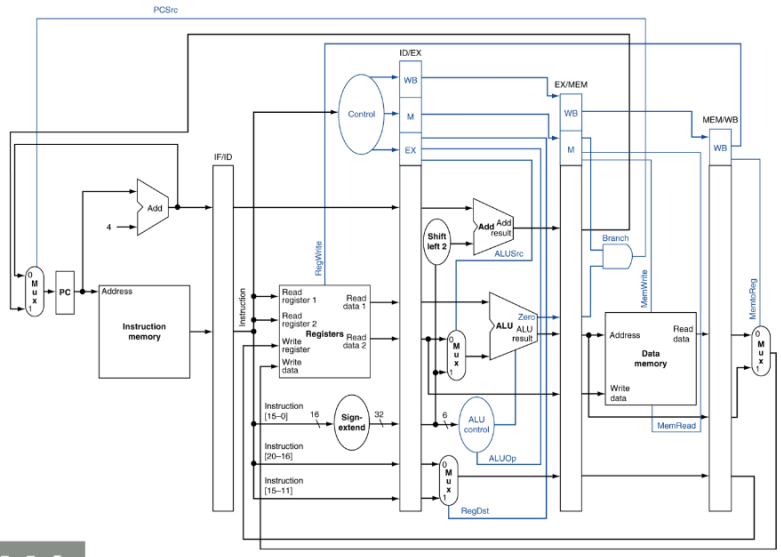


this write won't conflict as R/W happens at diff times, & W at those places happen only after MEM/WB.

the same datapath prblm - for R-type too...



Pipelined Control



WRITE CONTROLS LATER.

add $r1, $r2, $r3
add $r4, $r1, $r2

IF ID EX MEM WB
I_1
I_2 I_1
I_2 I_1 →
I_2 I_1
I_2 I_1

SOLVED BY FORWARDING

exec. with older $r1.