ORG

DR.WALEED DWEIK

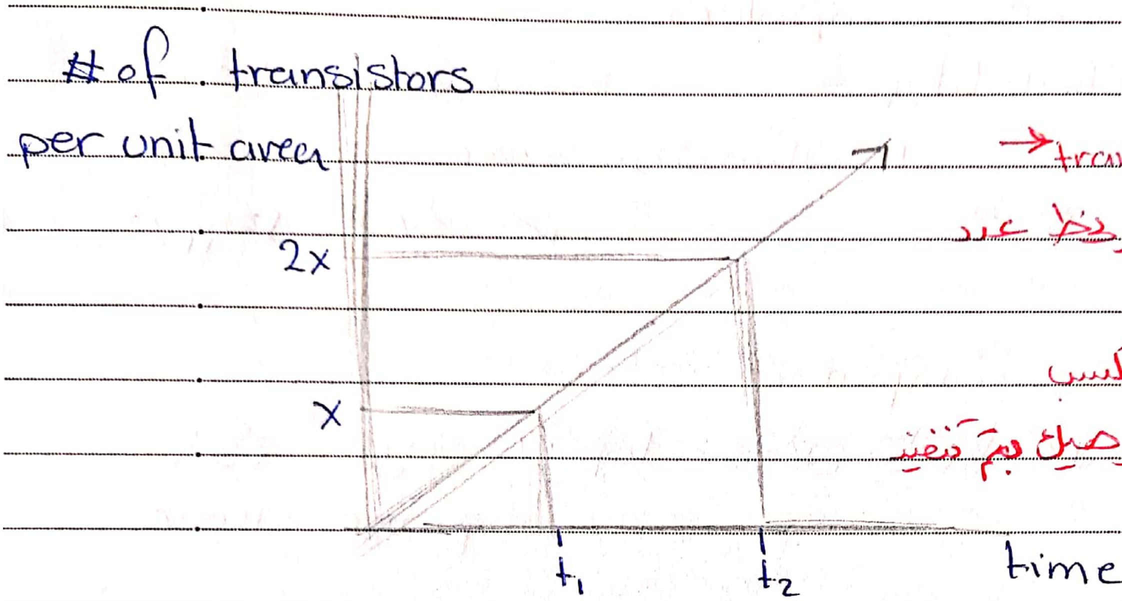
BY:FARAH HATEM

POWERUNIT

Scanned with CamScanner

The computer revolution

(Moore's Law)



← لما يكبروا عدد ال transistors →
ع نفس المساحة ، بنقدر نحط عدد
أكبر gates
← هيك بصير نقدر نعمل ونكسب
functionality أكثر وصار بنقدر ننفذ
multiple operatio
at the same ti

Gates → transistors

transistor → بنقدر نعتبره زي ال switch → وظيفته زي وظيفة ال switches

open ← بمرق التيار

close ← ما بمرق التيار

Classes of computers

→ Servers computers

Greliability → يعني انه اذا صار عندي أي مشكلة في ال server وأي مشكلة بال Hardware فيتم الحل عليها ✓ بنه

→ Super computers

بالموقع top500.org بيطلع قوائم بأخطر انواع computers هاي ال

↳ High-end scientific and engineering calculations : weather predictions, search Engines

فكرة الأساسية من ال Post PC Era وال cloud comp. هو انه جزء من ال software بكون بدل ما run عال DVD وجزء منه بدل ما run عال cloud

Scanned with CamScanner

[Eight Great Ideas]

① Design for Moore's Law

years for designing

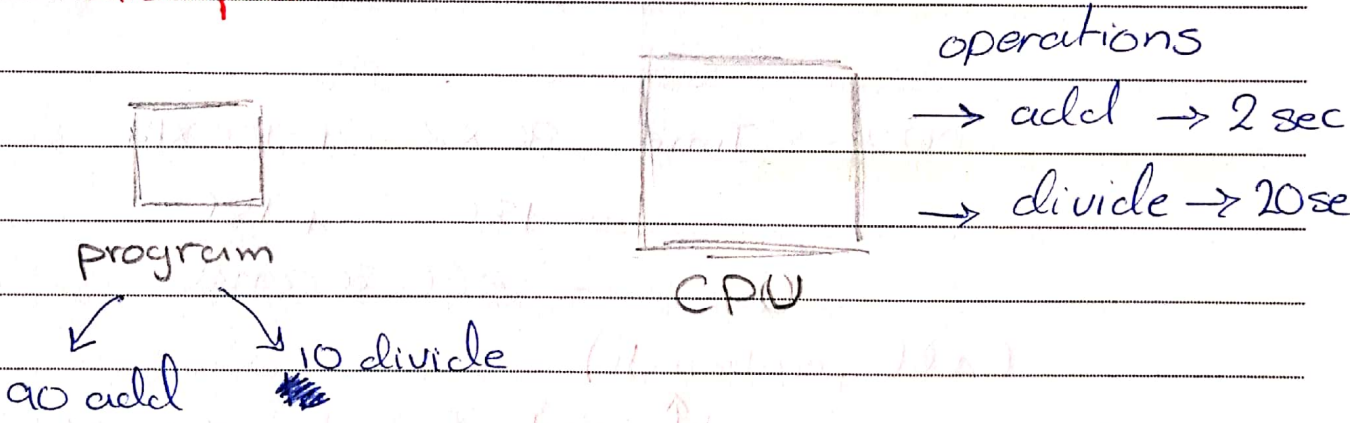
② Abstraction to Simplify Design

→ low level details hidden to offer simpler Model

→ Increase Productivity for Architects and programmers

③ Common case faster

Examples:-



CPU original ⇒ time = 90 x 2 + 10 x 20

= 180 + 200 = 380 sec

Scanned with CamScanner

Design Modified :-

↳ Add : 1 second

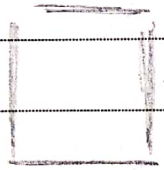
↳ divide: 20 seconds

CPU 1

CPU 1 → Time = 90 x 1 + 10 x 20

= 290 sec.

OR



↳ Add → 2 seconds

↳ divide → 18 seconds

CPU 2

CPU2 → Time = 90 x 2 + 10 x 18

= 180 + 180

= 360 seconds

(Add operation ال)

common case ↑ أحسن واحد فيهم، لأنه هو ال CPU 1 ← .

فنحسن جميع عمل ال CPU ، وعرفنا إن ال Add هو ال common لأنه بتنفذ بأكبر عدد مرات (90)

Scanned with CamScanner

← مش دايماً تحسين ال common case هو الأفضل، أحياناً
عملنا مرا يحسن ال rare case بطريقة تخلي عمل ال CPU
أفضل مما لو حسنّا ال common case، لذلك الشي غير ثابت

ومع ذلك بغالبية الحالة هي الأجود لتحسين ال common case
بشكل عام.

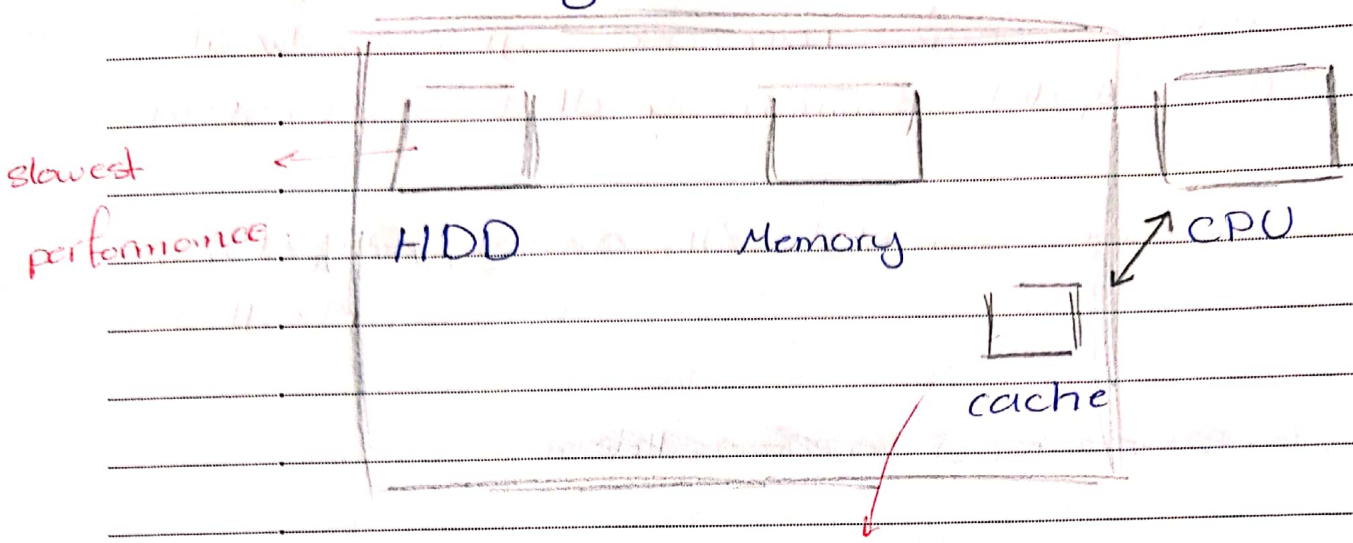
④ Performance via Parallelism

كل ما قدرت تحط resources أكثر بال CPU، وما كانوا
يعتمدوا على بعض، وصول ال resources كانوا يشتغلوا مع
بعض in parallel، بنقدر هيك نخلي تنفيذ لأكثر من برنامج
في وقت واحد، بصير ال performance أفضل.
← بنزيد عدد ال CPU's

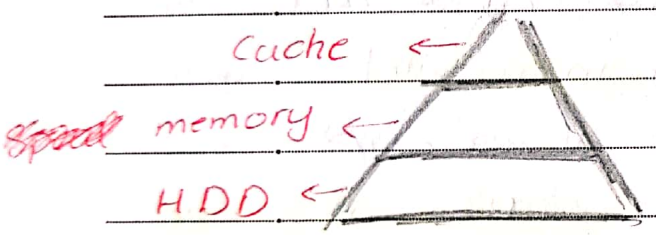
⑤ Performance via Pipelining
stages in CPU

⑥ Prediction (95%)
→ Based on History of performance.
Forgiveness better than permission.

⑦ Hierarchy



Most frequent Data saved in here.



size decreases

speed increases (time decreased)

cost increases

⑧ Dependability via Redundancy.

Or Reliable.

Scanned with CamScanner

High-level language

→ provides for productivity + portability

كتابة الكود بشكل أسرع ، وحجمه بكون أصغر

يوضع على أي Hardware.

The components of a computer

* five components →

The memory here is the main memory.

* HDD is considered a storage device.

* Network adapters is like (Network interface cards : NICs)

The Hardware support for graphics

← ال CPU في تلك لتخزين الداتا الخاصة بصورة معينة ولازم تتوضع لشاشة

↩ Frame buffer

← كل ما بدنا نعرض صورة جديدة القيمة اللي جوا هاد ال buffer بتتغير

Raster scan CRT display

المقدمة: بطريقة هاي مش موجودة بالسابقات سبب التوضيح لاسم:

[CRT] stands for "Cathode Ray Tube". and it is the technology used in traditional computer monitors and TV's. The image on a CRT display is created by firing electrons from the back of the tube to phosphors located towards the front of the display.

Inside the Processor (CPU)

→ Datapath → وكاشي بتمر فيه البيانات وبصير عليها عمليات فيه
Graelders, Decoders, Muxes

→ Control Datapath بتتحكم في ال

← ال cache بتكون جزء من ال CPU
(on chip) ↓

no delay in performance

Scanned with CamScanner

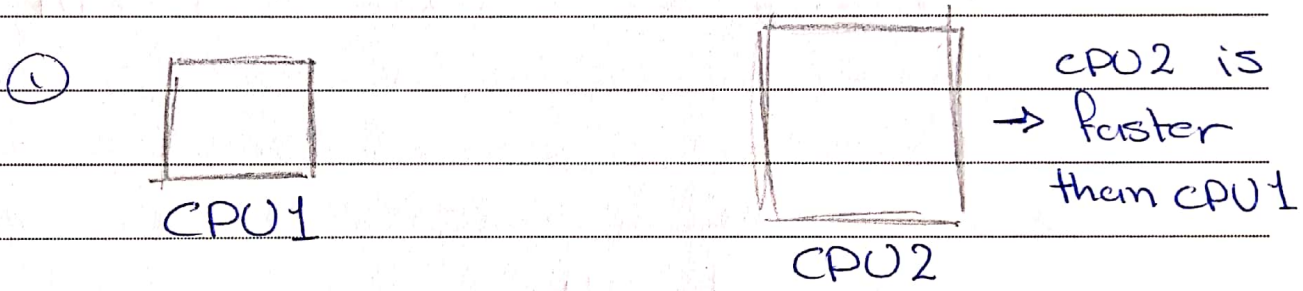
Solid State Drive (SSD) instead of HDD

ال SSD مصمم على نفس التكنولوجيا اللي مبنية عليها ال Flash memories

* أي اللي بخزن بيه بدل ما run لازم يكون متواجد بصورة رئيسية في ال DRAM

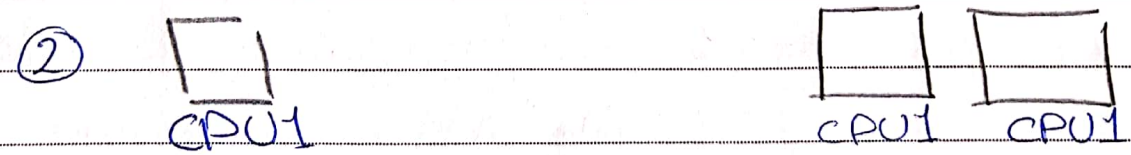
Response Time & Throughput

example :-

① 

→ Response time decreases

→ Throughput increases

② 

→ Response time stays the same, assuming that ~~instructions~~ tasks are can't be excuted in parallel between the 2 CPU's ← ما بتقسم على اثنين

→ throughput increases.

Scanned with CamScanner

في المثال الثاني ، ال response ظل زي ما هو لأنا افترضنا انه
ال task الواحدة ما بتنقسم على two CPU's

لكن ال throughput زاد لأنه بكل حالات في امكانية انه يتم تنفيذ
two tasks ، بالتوازي بين ال two CPU's

* دائماً ال response بزيد ، ال throughput بزيد
لكن لو ال " ثبت ، ال " مش شرط يثبت

Chapter 1
Section 1-4

(Summary &
Rearrangement)

Scanned with CamScanner

layers of computer

layer		levels of program code.
Application software.	- [HLL] - more natural - conciseness - programs independent from harelware.	- productivity + portability ✓ - closer to problem domain.
System Software.	- compiler (HLL to machine code) - Operating System (service code)	- Assembly language. - textual representation of instructions.
Hardware.	- processor - memory. - I/O controllers.	- Binary Digits - every thing is Encoded

Scanned with CamScanner

Components of A computer

Input + Output

Main Memory

Data path & control

① Input + Outputs (devices)

* ↳ Interface
  - Display, keyboard, mouse
* ↳ storage
  - HDD, CD/DVD, flash.
* ↳ Network Adapters.
  - [communications with other computers]

+ The LCD screen and Touch screen

→ LCD screen (Liquid Crystal Display)
↳ showing the bit map.

→ frame buffer restores the map
→ Refresh rate reads the bit pattern of pixels.
→ CRT display, displays the newest bit map

Scanned with CamScanner

→ TouchScreen.

two types → Resistive

→ Capacitive (multiple touches)

→ Network Adapters (NICs)

Networks

LAN (local area network) → Ethernet.

WAN (Wide Area Network) → Internet.

(Wireless Network) → Wi-fi, Bluetooth.

→ (2) Main Memory + Storage Devices

+ slide 21

	Volatility
DRAMs	✓ (DRAM): Dynamic Random Access Memory / holds data while running.
Magnetic Disks (HDD)	× → also like optical disks (CDROM / DVD)
Flash	×

Scanned with CamScanner

(3) Inside CPU (Processor)

→ Datapath

→ Control

→ Cache Memory nowadays is designed to be (on chip)

SRAM → Static Random Access Memory.
instead of slower and cheaper DRAM.

Scanned with CamScanner

§ 1.6 → Performance

في حالة ال CPU اهنا بدنا ال performance بدل ال execution او ال response time

علاقة عكسية بين الأداء وال execution time.

$$\text{performance} = \frac{1}{\text{execution time}}$$

performance (P)
Execution Time (ET)

$$\frac{P_x}{P_y} = n$$

better ← X → CPU 1
Y → CPU 2

↳ x is n time faster than y

$$\frac{P_x}{P_y} = \frac{ET_y}{ET_x} = n$$

A → 10 seconds B → 15 second
A is faster than B (1.5 times)

$$\frac{ET_B}{ET_A} = 1.5$$

Scanned with CamScanner

اذا بدنا نطلعها كنسبة ←

A is 1.5 times faster than B.

اطرح 1 من (n)، الي هو بمثل لسرعة نفسها واضربه بـ 100%

A is 50% faster than B

مثال

A is 2 times faster

2 - 1 = 1

1 * 100%

A is 100% faster than B

Scanned with CamScanner

Measuring Excution Time

عشان نعرف نحسب ال response time لازم نكون عارفين اصطلاحين مهمين system performance و ال CPU performance.

System performance: we are talking about memory, input ~~outpt~~ output devices, altogether
← من لما تبلش البرنامج لحد ما البرنامج يخلص بلك تحسب الزمن كامل

↳ this time is called: response time / Elapsed time / excution time / wall clock time.

→ performance ⟷ للسستم كامل
processing time + I/O + OS time + idle time

Idle time: الوقت الي بكون فيه الكومبيوتر ما بعمل اشي.
لما يكون البرنامج بعمل wait لما يكون ال OS ببدل بين ال programs بالدور.

$$\text{System performance} = \frac{1}{\text{Elapsed Time}}$$

CPU performance

the time here is called CPU time

Total → CPU time = processing time + OS overhead

User CPU time — System CPU time

System performance = $\frac{1}{\text{Elapseed T}}$

CPU performance or User CPU performance = $\frac{1}{\text{User CPU time}}$

من يوم وطالع لما ذهب سيرة ال performance بكون قصدنا عن ال User CPU performance ، لأنه كده راستاغنه

User CPU time:
CPU time spent on the program

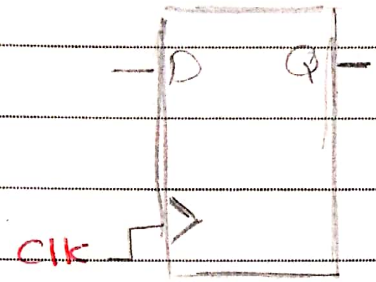
System CPU time:
CPU time spent in the OS performing tasks on behalf the program.

Scanned with CamScanner

CPU Clocking

remember CPU is → control
→ Datapath

→ Datapath → FlipFlops جزء كبير منها



FlipFlop

ال FlipFlop
بخزن ← 1 bit
كل ما يجي positive edge

الزمن بين ال cycles ثابت

clock period: (T)
duration of a clock cycle. (second)

clock Frequency (F), rate (R):
cycles per second (Hz)
1 Hz = 1 cycle / second

4×10^9 Hz → يعني ال CPU بعمل 4×10^9
عملية في الثانية أو task
مهمة.

Scanned with CamScanner

* اذا اعطاني مثلاً انه عندي CPU معين ال cycle ال clock الوحدة فيه بتاخذ 250×10^{-12} ثانية ، مطلوب ال frequency

حل:

$$\text{clock } f = \frac{\text{cycles}}{\text{seconds}} = \frac{1}{250 \times 10^{-12}} = 4 \times 10^{9} \frac{\text{cycle}}{\text{second}}$$

$$= 4 \times 10^{9} \text{ Hz}$$

example:-

$T = 0.5 \text{ ns} \rightarrow f = ??$

$$f = \frac{1 \text{ cycle}}{0.5 \times 10^{-9} \text{ second}} = \frac{1 \times 10^{9}}{0.5} = \frac{10 \times 10^{9}}{5}$$

$$= 2 \times 10^{9} \text{ Hz}$$

Rule :- $R = F = \frac{1}{T}$

and: $T = \frac{1}{F} = \frac{1}{R}$

example:-

when F = 2.4GHz, T = ?

$$T = \frac{1}{F} = \frac{1}{2.4 \times 10^9} = \frac{1 \times 10^{-9}}{2.4} \approx 0.41 \times 10^{-9} s$$

$$= 0.4ns$$

0.4ns for one operation, to change its state

User CPU time:-

user CPU time = CPU clock cycles * T

$$= \frac{\text{CPU clock cycles}}{\text{clock Rate}}$$

→ shorter user CPU time → better performance.

Scanned with CamScanner

* Performance improved by

① Reducing number of clock cycles needed for one task

② Increasing clock rate / Decreasing clock cycle time

example :-

$f = 4\,GHz$

$T = \frac{1}{4\,GHz} = \frac{1 \times 10^{-9}}{4}$ second

$0.25\,ns$

↓

when increasing frequency

$f = 8\,GHz \longrightarrow T = 0.125\,ns$

Scanned with CamScanner

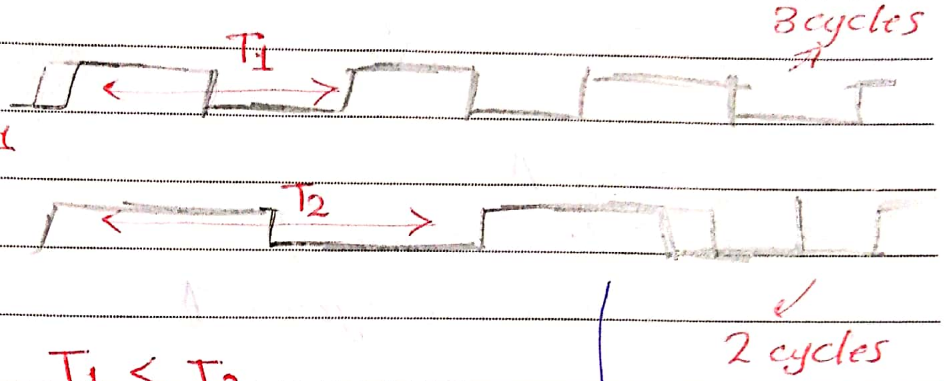
← دايمًا لما نحاول نطور نقطة ، وحدة منهم التانية راح تتراجع

③ Hardware designers must often trade off clock rate against cycle count

example:

task :
CPU1

CPU2



$T_1 < T_2$

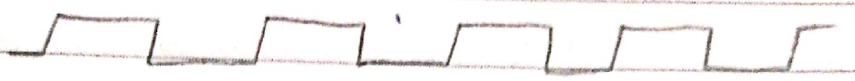
الحالة الثانية افضل لانه بالنهاية الزمن الكلي اقل.

CPU1 time $= 3 \times T_1$

CPU2 time $= 2 \times T_2$

Scanned with CamScanner

Example:

task: 

task: 

Total time ↓

Rate ↑

cycle duration ↓

number of cycles ↑

CPU time example:- (Slide 30)

faster clock → faster frequency.

Scanned with CamScanner

Example slide 30

Computer A	Computer B
User CPU time = 10s	User CPU time = 6s
Frequency = 2 GHz =	Frequency = ? ✓ (faster clock)

→ 1.2 clock cycles in computer B ✓

when asking for how ~~much~~ faster it has to be → find new frequency.

for A

CPU time = clock cycles / R ✓

$$10 = \frac{\text{cycles}}{2 * 10^9}$$

$2 * 10^{10}$ = cycles for A

for B

cycles = $1.2 \times 2 \times 10^{10} = 24 * 10^9$

$$\text{CPU time} = \frac{\text{cycles}}{f}$$

$$f = \frac{24 * 10^9}{6} = 4 \text{ GHz}$$

Scanned with CamScanner

always to determine which computer is better, you depend on its performance that depends on User CPU time.
(do not use (f) or cycle time to compare)

→ for the previous example:
How much B is better than A?

$$\frac{\text{Performance B}}{\text{Performance A}} = \frac{\text{CPU time A}}{\text{CPU time B}} = \frac{10}{6} = 1.667$$

B is 1.667 times faster than A
" " 66.7% " " " "

Finally
User CPU time → time needed to finish a task in second unit

Scanned with CamScanner

number of clock cycles:-

← هو عدد ثابت ، يختلف من برنامج لبرنامج لنفس ال CPU

Instruction count and CPI

Instruction count:-

عدد التعليمات الموجودة في البرنامج بيخلي اوعى ال task اللي بدي اسويها

← لما دخول البرنامج لـ machine code عن طريق ، للوصايل بين هنا كم instruction موجود عا بالبرنامج

← من مواصفات ال Hardware راح نعرف كل instruction كم cycle راح ناخذ

← هاي بنيها (cycles per instruction) CPI

Instruction counts (IC)

clock cycles = IC × CPI

User CPU time = IC × CPI × clock cycle time

$$= \frac{IC \times CPI}{\text{clock Rate}}$$

Scanned with CamScanner

← احنا بنفترض لما نحسب انه ال CPI ثابتة

← لكن اذا كان متغير

لـ

addition 3 cycles

subtraction 4 cycles

Multiplication 6 cycles

ومنشان هيك بناخذ ال average تبعهم

لتحديد → (CPI average)

→ ① Instruction count for a program.

→ affected by

Application Software ← *program.

Hardware ← * ISA : Instruction Set Architecture.

كل Hardware ال لغة مختلفة ، وحسب هاي اللغة

الي بفهمها بكون في مجموعة instructions مختلفة

عن ال Hardware الثاني

← حسب شو هي ال instructions الي بفهمها ال CPU

بكون بختلف عدد ال instructions الي راح تنفذ البرنامج

لما نحوله من HLL ← ل machine code

System Software ← * compiler

← لما ال كومبايلر يحول ال HLL ← machine code

ممكن يكون في واحد ~~ ~~ اذكى من الثاني

نحوله ل instructions اقل

→ Average cycles per Instruction

→ Determined by CPU Hardware

* If different instructions have different CPI (Average CPI affected by instruction mix)

↳ هاد شرط بدونه ما بتزبط المقارنة ✗

→ When comparing between two CPUs you need to give them the same program, considering the same ISA and compiler.

↳ Same Instruction count ✓

Example:- Slide 28

Computer A — Cycle Time = 250 ps

Computer B — Cycle Time = 500 ps

average → CPI = 2.0 cycle/Inst

CPI = 1.2 cycle/Inst ← average

دائماً بلزم عدد صحيح من ال cycles لأي instruction مش عادي يكون في كسور

لهيك بناخذ ال ← average

$$CPU\ Time_A = IC * CPI_A * Cycle\ Time$$
$$= IC * 2.0 * 250ps = IC \times 500ps$$

$$CPU\ Time_B = IC * 1.2 * 500ps = IC \times 600ps$$

A is faster $\frac{600}{500} = 1.2$ times than B → (20%)

CPI in more Detail:-

Two ways to evaluate CPU clock cycles:

① CPU clock cycles = $\sum_{i=1}^{n} (CPI \times IC_i)$

where your program has more than 1 Instruction type, you take then, the sum of all of them. (i is one of these different instructions)

② CPU clock cycles = $IC_{total} \times CPI_{average}$

IC here is:- total Instruction count, for all Instruction types included in the program

CPI here is: Weighted average CPI

Weighted average CPI

→ how to get the equation:-

User CPU Time = CPU clock cycles × cycle Time

$= \sum_{i=1}^{n} (CPI_i \times IC_i) \times T = IC_{total} \times CPI_{avg} \times T$

Then $\sum_{i=1}^{n} (CPI_i \times IC_i) \times \cancel{T} = IC_{tot} \times CPI_{avg} \times \cancel{T}$

Scanned with CamScanner

Then → $CPI_{avg} = \dfrac{\text{CPU clock cycles}}{\text{Instruction count}_{total}}$

$$CPI_{avg} = \frac{\sum_{i=1}^{n} (CPI_i \times IC_i)}{IC_{total}} = \sum_{i=1}^{n} \left(CPI_i \times \frac{IC_i}{IC_{total}} \right)$$

✓ Relative frequency.

← بغض النظر عن الطريقة اللي بنختار بها الحل ١ ، ٢ الجواب ح يطلع نفسه لل User CPU time.

if you are given that, "they are running on the same hardware", while comparing between two computers?- you know that:

1. Same clock period (same clock rate)
2. Same CPI_i of i (of one instruction, not the average)
   ↳ the average may be different due to the number of sequence of the instruction

→ Same ISA + Same Compiler = 1. Same Instruction count total
(And not vice versa)
2. Same IC_i for i

Example / slide 31 //

هو ما عدلنا ال cycle time , وما حكالي الأمر نفس ال hardware . لهيك ما بنقدر نعرف مين فيهم الأسرع .

Example / slide 32 //

→ Two compiled sequences :
two different compilers , compiler 1 , compiler 2

→ running on the same computer, (same hardware)
↳ Same CPI_i of the two sequences (معطينا بالجدول)
↳ Same clock cycle time
↳ Same ISA → ما بستفيد منها هاي المعلومة للسؤال
compilers are different.

Solution:-

→ Given that : $\frac{ET_2}{ET_1} = 2 = \frac{\frac{\text{CPU clock cycles}_2}{\text{clock rate}}}{\frac{\text{CPU clock cycle}_1}{\text{clock rate}}}$

$= \frac{\text{Cycles}_2}{\text{Cycles}_1} = 2 = \frac{\left(\sum_{i=1}^{n} CPI_i \times IC_i\right)_2}{\left(\sum_{i=1}^{n} CPI_i \times IC_i\right)_1}$

$2 = \frac{8+2f}{17} \Rightarrow f = IC_B = \underline{13}$ count for B needed.

Example-

→ Two compiled sequences

→ relative frequencies for IC_i are given in a table

→ on different hardwares

Seq1 → CPU1 → $T_1 = 2ns$

Seq2 → CPU2 → $T_2 = 1ns$

	A	B	C	D	IC_i
Seq1	20%	10%	50%	20%	
Seq2	0%	40%	30%	30%	

	A	B	C	D
CPU1 ← CPI_i	1	2	3	4
CPU2 ← CPI_i	3	1	2	6

$CPI_{avg_1} = 1 \times 0.2 + 2 \times 0.1 + 3 \times 0.5 + 4 \times 0.2 = 2.7$

$CPI_{avg_2} = 3 \times 0 + 1 \times 0.4 + 2 \times 0.3 + 6 \times 0.3 = 2.8$

Scanned with CamScanner

Chapter 2: Instructions : Language of the computer

↓

ISA

2.1 / Introduction

Computer languages → with simple Instruction sets

→ with complex Instruction sets

~~2.2 Architecture~~

2.2 / Operations of the computer hardware

one arithmetic instruction → 3 operands → one operation

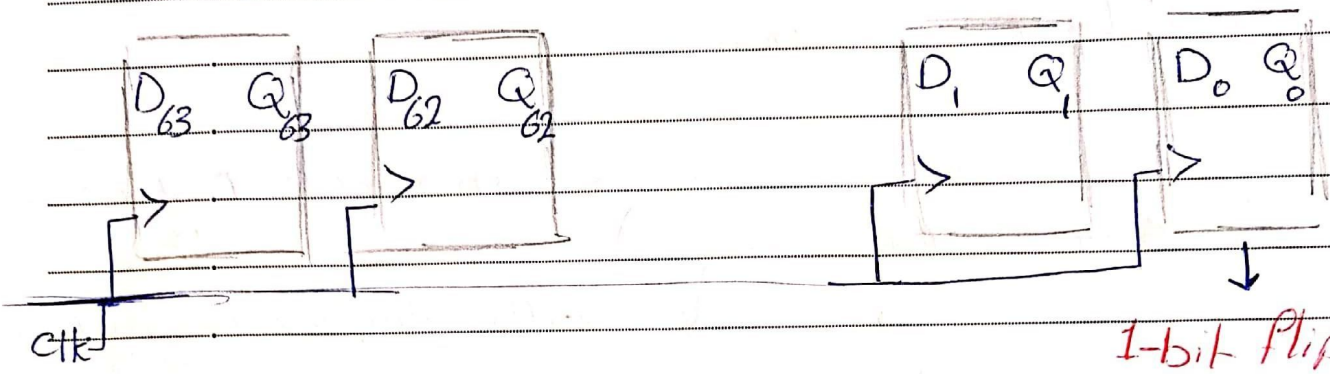
(for the simplicity of IS)

3 operands → 2 sources, 1 distination / 1 operation

an operand can be a source and a distination in one instruction

Scanned with CamScanner

→ Register file = group of registers

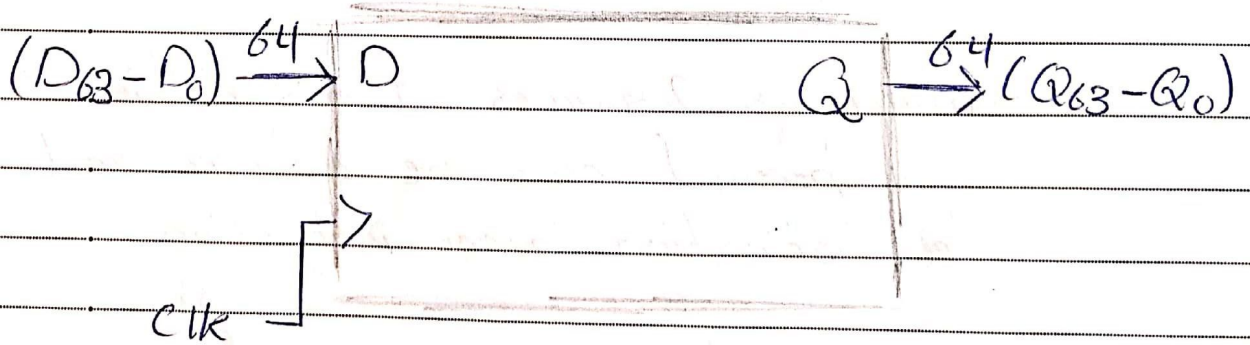


1-bit flipflop

1-bit register

$(Q_{63} - Q_0)$ source

$(D_{63} - D_0)$ write on distination



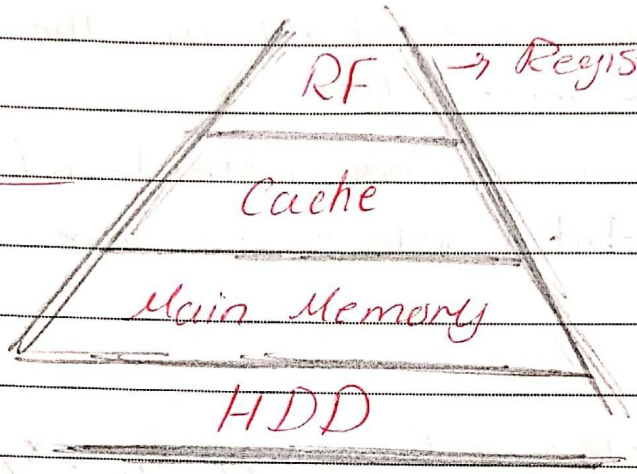
64 bit register

Scanned with CamScanner

الاختراع لل Risc-V انه يكون معاه هادا register

→ 32 registers in the register file

↳ Risc-V has ~~32x64~~ 32×64-bit register file



the new hierarchy

~~[illegible]~~

→ Arithmetic instruction only use operands saved in (RF)

معناتي اني بمنع انه نستخدم اي register الا اللي احنا بدنا اياه.

رغم انه بعضهم الهم استخدامات تانية.

→ Register file used to save most-frequent data, (frequently accessed data)

→ هاي ليست الي استخدمها اول، بتكون بال cache او ال MM

← بال arithemetic operations كل ال operands بتكون register

→ Risc-V has a fixed length of 32-bit for ~~each~~ instructions

→ instructions are stored in the memory

One register File:-

big register ← - 32-bit for instructions (the length of them)

- 32×64-bit registers to store data

→ Example: add x_5, x_6, x_7 — Assembly

x_5 → distination, x_6 → source 1, x_7 → source 2

مثال 32-bit ، تكون ال instruction ، لازم يكون في مجموعة bits

بتحكي للجهاز مين هو ال distination ، ومين ال source 1 ....

distination ↓ رقم 5 → 101

source 1 ↓ رقم 6 → 110

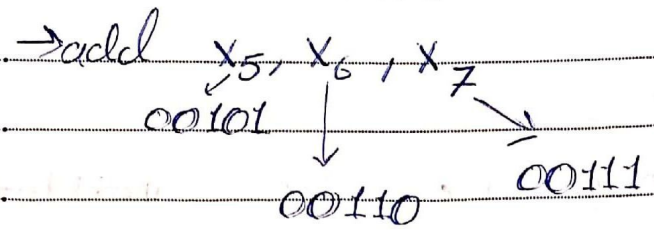
← بحدد عن طريق ال binary bits يقابلها لكل رقم ← source 1 = 6 = 110

source 2 = 7 = 111

وهنا 32 ← register ، أكبر رقم بتخزن جواهم 31 = 11111

يعني بدو 5 ← bits ، فبنخصص لكل register ← 5-bits

حتى نحدد هو وين داخل العملية



→ for each instruction you use 3 registers in the register file

Risc - V

- **Data**
  - 64-bit (doubleword)
  - اشي بنخبيه بتخزنه في ال RF واشي في memory
- **Instructions**
  - 32-bit length
  - memory بتخزنوا في ال

Scanned with CamScanner

RISC-V registers:-

X_0: the constant value 0

← لقيوا انه ال computer بيحتاج احياناً يكون عنده رقم 0 مخزن جاهز.

~~temp~~

→ temporary registers

→ you can ~~over~~ over write them any time

→ saved registers

→ you can not over write them

→ for memory operands:-

→ Read / load from memory → register

→ store / writer on memory → from register.

Loading from memory to ~~my~~ register and vice versa, is copying data, not changing its place completely.

Scanned with CamScanner

→ when you read from memory, you give it the address of the first byte of double word.

→ Memory organization

RISC-V little Endian:-

Example:-

4 bits

M[0] = 0X F132 BA5C AD34 7900

long long int — 16 bit. — most significant bit — least significant bit

this doubleword takes 64-bit (16×4)

9		
8		
7	0X	F1
6	0X	32
5	0X	BA
4	0X	5C
3	0X	AD
2	0X	34
1	0X	79
0	0X	00

bytes in memory.

→ Little Endian

Big Endian

0X 00

0X 79

0X F1

Scanned with CamScanner

Data transfer instructions:-

→ address of M[i] = starting address of M + the offset ← index (i) × array type size in bytes

the element of an array

example:- ~~long~~ long long int: array type size = 8 bytes.

address = starting address + i × 8 bytes.

→ load as an instruction:-

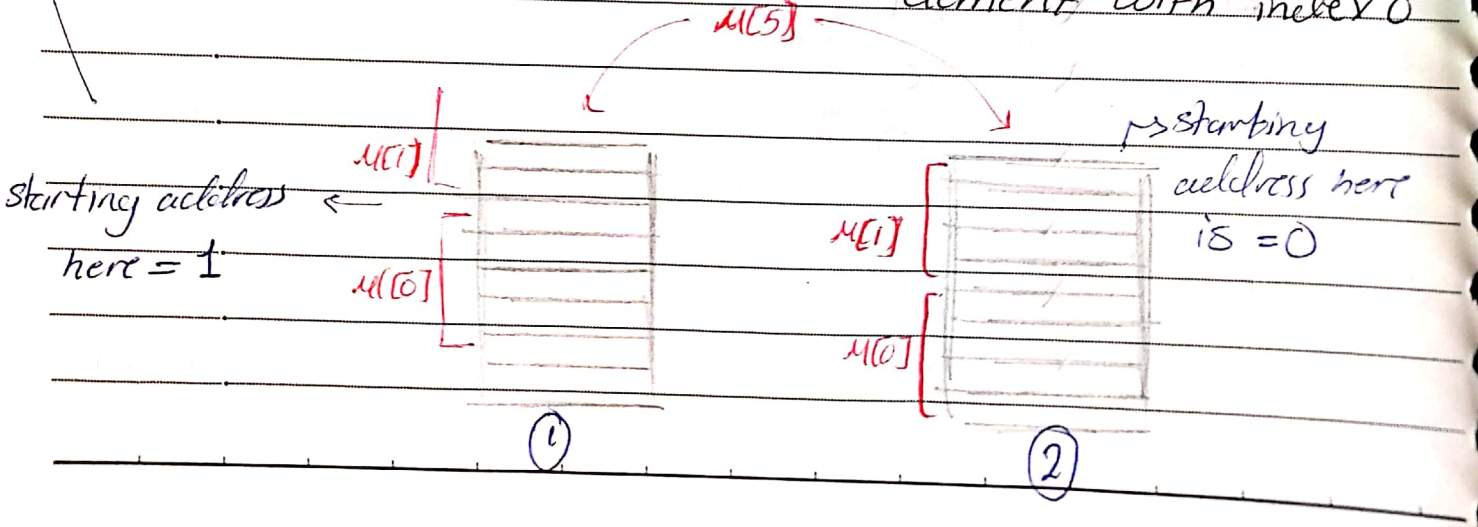
← بتحدد وين هوا ال memory location اللي بدي اعمل منه كوبي للداتا.

← لازم كمان حدد وين هوا ال register اللي بدي احط الكوبي هاي و أرجعلي

the form of load doubleword instruction:-

Ld destination register , offset(base register)

→ Starting address of array ≡ address of the element with index 0



Scanned with CamScanner

Example:- ✓ على المسألة رقم ② في الصفحة اللي قبل

address of M[2] = 0 + 2x8 = 16

address in memory.

remember :-

offset = $index_{(i)}$ x array type size in bytes.

offset ← يعني زيادة / قديش بدي ارجع عن ال starting address

base register → هو ال register من ال 32 واحد
بكون مخزن فيه ال starting address تبع ال array اللي بدنا اياه

When writing this instruction, what happens?

example: ld x28, 32(x18) ≡ (x28) ← Memory [32 + (x18)]

① determining memory address
[address of M[i] equation]
32 + (x18)

② access memory at [32 + (x18)]

③ copy to the destination register (x28)

Example:-

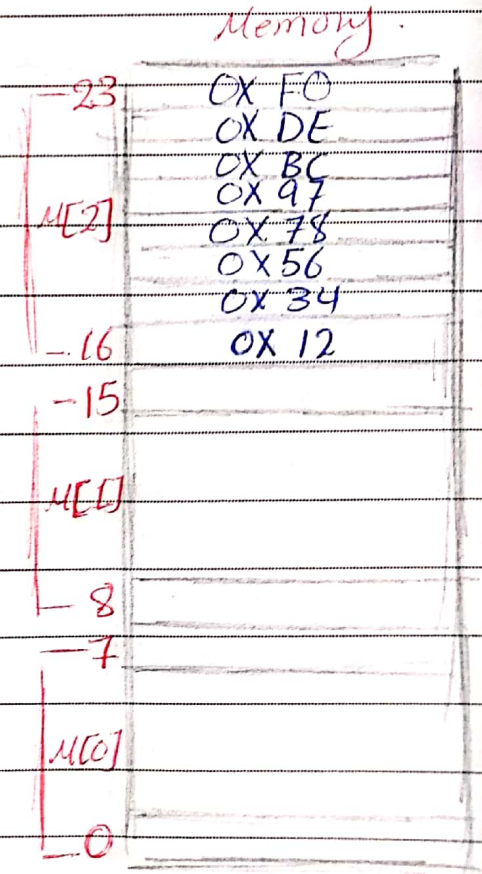
→ you are required to copy element M[2], to register (x25) in register file

→ ld x25, 16(x0) ≡

Memory [16 + (x0)]

(x25) ← Memory [16]

address of M[i] equation



Scanned with CamScanner

now x25 has the content: [0X FODE BC ---- 12]

→ When you want to copy and read data from an absolute address:

example: read memory content of address 5, and save it in register X_{10}?

Ld X_{10}, 5 (X0) → starting address = 0 always

↙ offset is absolute address

5 + 0 = 5 ✓

Scanned with CamScanner

register

→ Store as an instruction:-

Scl source register, offset(base register) ≡

Memory[offset+(base)] ← (source)

effective address

example:- sicle 13

g = h + A[8];

g in → x20
h in → x21
array A → starting address in x22

↳ better to choose one of temporary registers

ldr x9, 64(x22)
add x20, x21, x9

Scanned with CamScanner

example/slide 14

A[12] = h + A[8] -> in x5

-> h in x21

-> Array A -> starting address in x22

offset for A[12] = 96

" " A[8] = 64

ld x5, 64(x22)

add x6, x21, x5

sd x6, 96(x22)

-> element at right side of equation -> load

-> " " left " " " -> store

Scanned with CamScanner

→ Register optimization

هو الاعتماد قدر الإمكان على ال RF ، واللجوء بعد ذلك لل memory

→ immediate operands

addi reg, reg, constant ⟵ هيك الترتيب

← بصير الحجم محدود لانه ضمن ال instruction

← كل instruction ممثلة باستخدام 32-bit في ال memory
فمن هدول ال bits ، بكون موجود ال constant

→ no subtract immediate instruction

→ you are allowed to use negative constants

← رغم هيك ال constant الي مسموح تستخدمه محصور لعدد
معين.

Constant zero:-

→ useful to copy a value from a register to another.

~~→ to negate the value~~

addi x_{22}, x_0, x_{21}

immediate instruction works here:
addi: x_{22}, x_{21}, 0

→ to negate the value in a register.
sub x22, x0, x21

→ immediate instruction does not work here:
it doesn't working with the order.

Scanned with CamScanner

→ Binary Integers

value = $d \times r^i$
- d ← the bit
- r^i → weight

d range is $0-(r-1)$

→ if n = number of bits
then the range of that number =
$0-(2^n-1)$

min = 0
max = $2^n - 1$ ⇐ talking about combinations

for RISC-V :-
→ 2^{64} combinations in range $(0-(2^{64}-1))$
→ Bit 0 is the least significant for $(2^{64}-1)$
→ " 63 " = most " " "

Scanned with CamScanner

→ Signed Integers → 2's Complement representation

تحويل بين binary , decimal :-

$$X = -X_{n-1}2^{n-1} + X_{n-2}2^{n-2} + \dots\dots$$

هاد هو القانون
بين الأقواس الحادية وهاد النظام
اذا كان الرقم موجب راح يضل
القيمة الحقيقية للرقم
واذا كان سالب راح يتحول
القيمة الحقيقية للرقم ع الطرح

numbers :-
domain →
$max = +2^{n-1} - 1$

zero

$min = -2^{n-1}$

for n bits number

Example:-

→ 4-bit unsigned (0 - to - 15)
→ 4-bit signed (-8 → to → +7)

Scanned with CamScanner

2's complement characteristics.

→ in the 2's complement format, it's almost similar when solving for signed and unsigned numbers

[illegible]

*Imbalance between positive and negative numbers (the number of the positive numbers in the domain is lower)

*overflow:- is needing more bits to represent the number resulting when adding or subtracting two numbers of less number of bits.

← كال over flow في ال 2's complement بيصير ال:-

incorrect sign bit. ←

① (positive) + (positive) = negative result ✗

② (negative) + (negative) = positive result ✗

Signed Negation in 2's-complement

or finding the 2's complement for a number

negation → $x \rightarrow -x$

positive → negative

negative → positive

change the sign bit.

(1) first approach

do the one's complement, then add 1 to the number

(2) second approach

start from the first bit of value 1

→ Why 2's complement is called like this

$x + (-x) = 1000\cdots00_2 = 2^n$

n = number of bits of x

(2's complement = negation) of x

Scanned with CamScanner

- لما نكون في رقم عدد ال 1's غير كثير بنحوله من (X) لـ (-X)
او العكس باستخدام ال 2's complement عشان يصير
عدد ال 1's اقل ويصير تحويله ال decimal اسهل.

§ 2.5: Representing Instructions

compiler Assembler
HL → assembly → machine code.

Reduced instruction set computer (RISC-V)
→ Small number of formats for regularity / simplicity: High performance of low cost.

one instruction: 32 bit (32 ~~bit~~ register) in binary / 8 digits in hexa.

→ the instruction will be held in the memory, taking 4 bytes (the memory is byte addressable

Scanned with CamScanner

RISC-V R-format instructions

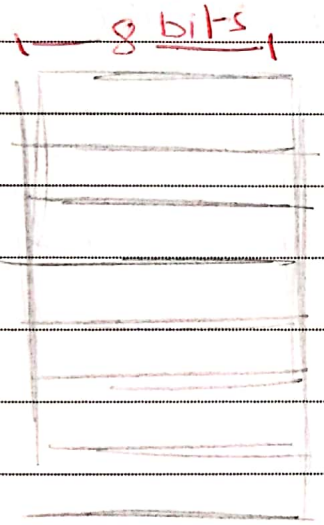
R for register

→ This format is used by 3 operands instructions

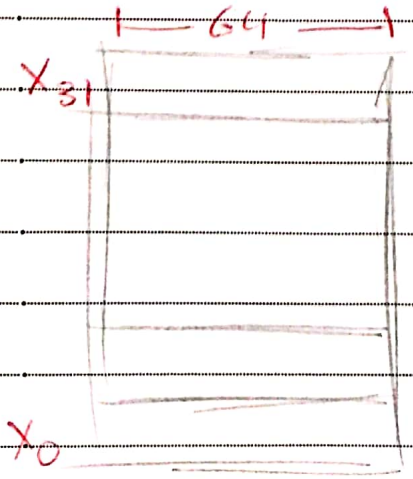
Remember

The width of the memory is 8 bits
byte addressable. ← (1 byte)

byte ~~address~~



the width of (RF) = ~~16~~ 64 bits (8 bytes)



Scanned with CamScanner

RISC-V I-format instructions

addi

(rd) = (rs1) + sign-ext (imm)

64-bit + 12-bit

↓ extend to 64 by 2's complement sign extension.

The range of immediate:

-2^{11} → to → $+2^{11}-1$

→ The offset can be negative, and that is when you go down in the memory.

→ The range of the offset is similar to the range of immediate.

$(\pm 2^{11})$ لكن هو للتعريف بشال ال (-1) و ع ل

but you need extension.

Memory address = sign-ext (offset) + baseaddress

Scanned with CamScanner

Binary compatibility : the same machine code can work on different Hardwares

→ This means that these computers use same ISA's

→ forward compatibility and not backward.

→ Data stored in the memory : 64-bits

→ instructions : 32-bits

Instructions :

① → and → R-format

andi → I-format

→ Bit by Bit Not → ~

Scanned with CamScanner

logical operations

@ Shift Immediate

→ Shift operations that are mentioned in here are logical not arithmetic

shift left ← → shift right

→ slli rd, rs1, imm
→ srli rd, rs1, imm

imm = number of bits to be shifted

shift left:-
slli rd, rs1, imm =

$(rs1) \times 2^{imm} = (rd)$ → كل حد مضروب في 2 للقوة اللي جنبها imm

example:-
slli $x_5, x_6, 2$ → $x_6 = 0 \cdots 00011$
$x_6 = (3)_{10}$

00 ... 000011 ← 00

after shifting → $x_5 = 0 \cdots 0001100$
$= (12)_{10} = (3 \times 2^2)_{10}$

the value of x_6 has not changed.

Scanned with CamScanner

*The maximum immediate you can use to shift is (64), using 64 zeros

→but it meaningless because:-
using 64 to shift left makes the resulting number = 0, while there are other ways to store (0) in the register

→so (63) make more sense (111 111)
max imm = $(63)_{10}$ = $(111\ 111)_2$: 6 bits you need for the max imm

based on that i

funct6	immed	rs1	funct3	rd	opcode
6 bits.	6 bits	5bits	3bits	5bits	7bits

till here it is similar to I and R formats

Scanned with CamScanner

Shift right :-

→ Fill with zero this direction.

→ For (unsigned only) :-

$(rd) = \frac{(rs1)}{2^{imm}}$

otherwise, shift right is not used for division for signed numbers.

example :-

srli x5, x5, 3

(x5) = (0 ... 000 10000)

$(x5) = (16)_{10}$

000 → 0 ... 0001 0000 $= (16)_{10}$

new ↓ out.

0 ... 0000 0010 $= (2)_{10}$

$\left(\frac{16}{2^3}\right)_{10} = (2)_{10}$

Scanned with CamScanner

RISC-V design principles:-

① principle ① : Simplicity favors regularity
→ talked about it when we mentioned arithmetic operation ~~with~~ with (3 operands) ; 2 sources , one distination

② principle ② : smaller is faster
→ ~~register~~ register file of 32 bit

③ principle ③ : Good design → good compromise

→ Attention:
Left shifting also affects signed numbers for large positive and ~~large~~ small negative numbers

← اذا الرقم كان (+) وصغير، واضربناه برقم صغير، راح يظله (+)
← اما اذا كان كبير، ممكن نغير اشارته بسهولة.

Scanned with CamScanner

for not immediate shifting:

sll rd, rs_1, rs_2

srl " , " , "

funct7	rs_2	rs_1	funct3	rd	opcode

↳ R-format.

sll x_5, x_6, x_7

0000 --- 00011 ← 0000

$3 \times 2^4 = 48$

$x_5 = 0 \cdots 000110000$

$x_6 = 0 \cdots 000011$

$x_7 = 0 \cdots 001000$

Scanned with CamScanner

Arithemetic shift (uses R-format)

→ a stands for arithemetic

sla rd, rs_1, rs_2

sra rd, rs_1, rs_2

← ايام ال logical shift . كنا نقول ١، ٢، ٣ الي بتدخل (0)

But for arithemetic
it checks for the sign bit and
makes shifting with it

example:-

sra x_5, x_6, x_7

x_5 = 111 - - - - - 11

x_6 = 1111 - - - - - 11 → sign bit = 1
shift to right by 1
using 1

x_7 = 000 - - - 001

→ And operations (masking)

① andd : R-format

andd rd, rs_1, rs_2

← بتاخد كل bit مع الـ bit اللي مقابل الحاويتل logical and

② andi : I-format

andi rd, rs_1, imm

imm = 12 bits and rs_1 is 64-bits
so you need to do sign extension

→ OR operations — زي and بس or

→ XOR operations

$1 \oplus 1 = 0$	$1 \oplus 0 = 1$
$0 \oplus 0 = 0$	$0 \oplus 1 = 1$

→ Invert some bits, leave others unchanged

→ when immediate is entirely full of ones the result is the opposite of the register.

anything $\oplus$ 1 = $\overline{\text{anything}}$

Scanned with CamScanner

→ XOR imm, is usual to implement (not)

xori x9, x10, 0xFFF

↓

it inverts the content of x9

Scanned with CamScanner

Compiling if statements:

C code:-

```
if (i==j)  f= g+h;
else f= g-h;
```

compiled code:-

* first way of coding:-

```
    bne  x22 , x23 , Else      → x22=i,  x23=j
    addl x19 , x20, x21
    beq  x0, x0 , Exit
Else: sub x19, x20, x21
Exit: . . . . .
```

first scenario of implementation:

→ bne : is true

↳ skip line of (addl) and line of (beq)

→ implement (Else) line, then complete to Exit sequentially.

other scenario:

→ bne : is false / never go to Else line.

↳ implement (addl) line sequentially.

→ to avoid implementing line of (Else) so that the implementation is logically false →

Scanned with CamScanner

→ implement line of (beq), to skip line of Else and implement line of (Exit)

→ instruction of (beq) is put to cover implementing, addl and subtract together (only one should be implemented)

The other way of coding starting logical implementation with (beq)

code in RISC-V :-

```
beq x22, x23, if   → label name line of if branch.
sub x19, x20, x21  → line for else branch implementation
beq x0, x0, Exit
if : addl x19, x20, x21
Exit: - - - ——
```

Scanned with CamScanner

Compiling loop statements:-

instructions: بتحكم بيها (bne) ← استخدام الـ

⇒ implementing: C code:-

```
while (save[i] == k) i += 1;
```

using: beq

```
loop: slli  x10, x22, 3
      add   x10, x10, x25
      lw    x9, 0(x10)
      beq   x9, x24, L1
      beq   x0, x0, Exit
L1:   addi  x22, x22, 1
      beq   x0, x0, loop
Exit: ---
```

Scanned with CamScanner

More conditional operations:

signed comparison:
- blt — Branch if less than
- bge — Branch if greater than or equal

(implement next code using blt)

c code:

```
if (a>b) a+=1;
```

```
a in x22 , b in x23

    blt x23, x22, continue
    beq x0, x0, Exit
continue: addi x22, x22, 1
Exit :  - - - -
```

عادةً استخدام الشرط الي يؤدي لحالة (else) او خلي الحالة مكتوبة بالكود false ، يعني عدل instructions اكثر

→ bne / and beq, dont care about, the sign of numbers, they stay the same instruction for both sign and unsigned numbers.

→ blt, bge care about that

we use bltu, bgeu for unsigned numbers:

bltu : branch less than unsigned.

bgeu : branch greater than or equal unsigned

→ working with memory addresses you can use (signed) or (unsigned) instructions of those, it never alters

Scanned with CamScanner

Bounds check shortcut.

array size = n

save[i]

$0 \leq i < n$ } this should be true to avoid out of Bound exception.

$0 \leq i \leq n-1$

Basically, this implemented (using) using these instructions:-

knowing that n is in (x11)

i is in (x20)

```
    bge  x20, x0, L1
    beq  x0, x0, OOB
L1: blt  x20, x11, L2
    beq  x0, x0, OOB
L2: ~~~~~~ - - - -
```

all are replaced by one line:-

bgeu x20, x11, IndexOutOfBound

if x20 >= x11 or x20 < 0 go to OOB

Scanned with CamScanner

Example:-

```
for (i=0 ; i<10; i++)
    save[i] = save[i] * 2
```

① i → (x20)
② starting address of save array → (x21)
③ array save is long long int

```
    addd x20, x0, x0
    addi x5, x0, 10      <- this line to put 10 in a register (x5)
loop: bge x20, x5, Exit
    slli x6, x20, 3
    add x6, x6, x21
    ~~ld x7, x7, 1~~
    ld x7, 0(x6)
    slli x7, x7, 1
    sd x7, 0(x6)
    addi x20, x20, 1
    beq x0, x0, loop
Exit: ---
```

→ x5 = 10
→ x6 = address

28 march

Scanned with CamScanner

Compiling loop statements (case/switch)

first way of writing them is RISC-V

switch	nested else if	RISC-V
switch(n)	if(n==0){	bne x20,x0,L1
{	:	block of instructions
case 0: ---;	:	case 0
---;	}	:
break;	else if(n==1){	beq x0,x0,Exit
		L1: addi x5,x0,1
		bne x20,x5,L2
case 1: ---;	:	block of instructions
---;	}	case 1
break;	else if(n==1){	beq x0,x0,Exit
		L2: addi x5,x0,2
case 2: ---;	:	bne x20 x5,Else
---;	}	block of instructions
break;	else {	case 2
		beq x0,x0,Exit
default: ---;	:	Else: default
---;	:	case instructions
---;	:	Exit: --
}	}	-- --

Scanned with CamScanner

Basic Block:

sequence of instructions, that they will be implemented no matter what standing for those conditions:-

- No branches except at end
- No branch target (label) except at begining

Scanned with CamScanner

§2.8 Procedure calling

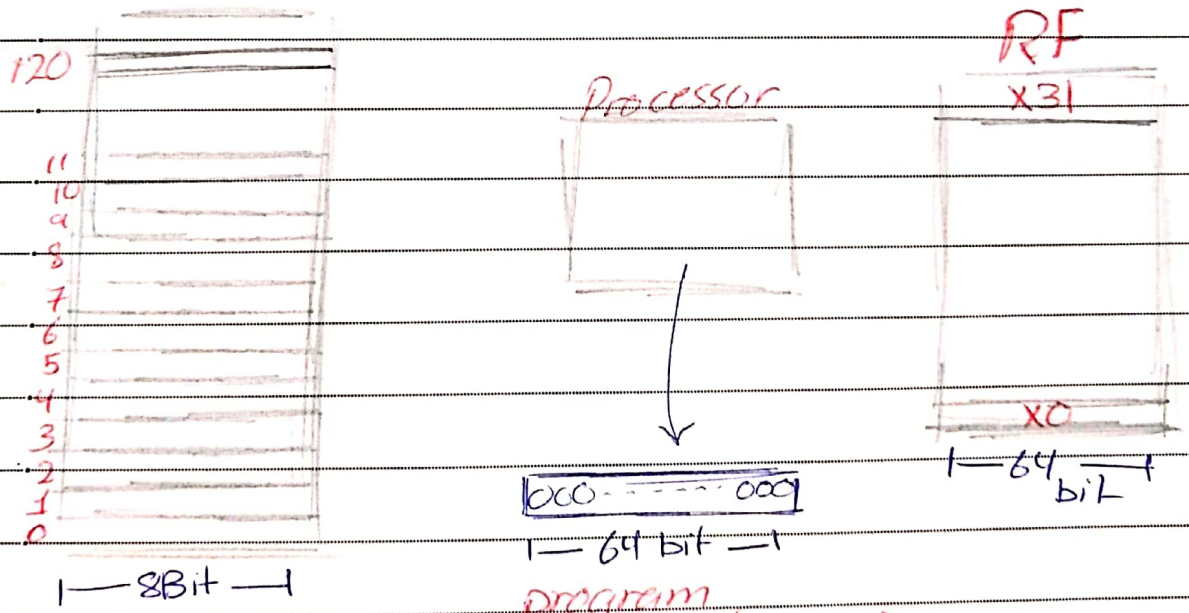
procedure → function / method

procedure call:-
jump and link (jal)

procedure returns
jump and link register (jalr)

→ jal, jalr change the value of PC.

what is PC? How it changes?



each instruction takes 4 bytes

(↳ it takes next instructions sequentially by (PC+4=PC) then takes the instruction that refers to the order of current PC

Scanned with CamScanner

How jal and jalr work:-

RISC-V

PC

0 main: - - - -

4 - - - -

8 - - - - - - jal x1, procedurelabel

12 - - - - - → call coverage (jal)

16 - - - - - jal do two things:-

20 ① (rd) = PC + 4

linking ← - (rd) = 16 we'll need it

jump ← ② changes PC to 120

120 coverage: - - - -

124 - - - - - -

128 - - - - - jalr x0, 0(x1)

132 - - - - - → return to main (jalr)

134 - - - - - jalr do two things:-

① (rd) = PC + 4

(rd) = 136 we'll not need it

② changes PC to 16

Scanned with CamScanner

jump and link (jal) → unconditional control instruction

jal rd, label

↓

x1 ≡ return address register

① linking → (rd) = PC + 4 → address of instruction currently executed.

② ~~jump~~ jump to a label → PC = address of instruction of label

jump and link register (jalr) → return from procedure.

jalr rd, offset(rs1)

① linking → (rd) = PC + 4 → in return is not needed so x0 is used as (rd)

② jump to instruction of address
↳ offset + (rs1)

PC = address = offset + (rs1)
= 0 + (rs1) = 0 + x1

Scanned with CamScanner

jal for control:

jal x0, label ≡ beq x0, x0, label

Using more registers: Stack

→ before execution, stack is empty.

→ Stack pointer

pointing at the top of stack, and it is (x2) one of general purpose registers. ~~(pointing at next empty location (not to fill))~~

(pointing at element, which is the last one before the first empty location)

Scanned with CamScanner

→ you should take care of registers used for each operation

C code:-

```
long long int leaf_example(
long long int g, long long int h,
long long int i, long long int j) {
long long int f;
f = (g+h) - (i+j);
return f;
}
```

name

arguments

Body

return statement

used registers

g → x10 f → x20

h → x11 temporaries x5, x6

i → x12

j → x13

In general you can use:

- (x10 - x17): registers used for argument, 8 registers.
- return value you can use x10, or x11.

Scanned with CamScanner

→ single procedure.

Leaf Procedure Example

- RISC-V code:

```
leaf_example:
  addi sp,sp,-24      shifting SP down     Save x5, x6, x20 on stack
  sd   x5,16(sp)      → SP is used as base address.
  sd   x6,8(sp)       → you can save registers in the order of your choice
  sd   x20,0(sp)
  add  x5,x10,x11     x5 = g + h
  add  x6,x12,x13     x6 = i + j
  sub  x20,x5,x6      f = x5 – x6
  addi x10,x20,0      copy f to return register
  ld   x20,0(sp)      Restore x5, x6, x20 from stack
  ld   x6,8(sp)
  ld   x5,16(sp)
  addi sp,sp,24
  jalr x0,0(x1)       Return to caller
```

(3.) save on stack

(4.) procedure Body

restore from stack

(6.) return

Notes:-

→ any register other than parameters, and return register should be saved on stack (in this example x20, x5, x6)

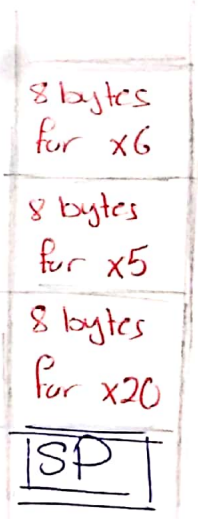
because each register is 64 bit sized, so, they will need for each one 8 bytes.

→ 3 x 8 bytes

→ = 24 byte

so (SP) will be shifted down 24 bytes

This is a subtraction of the value of (SP) by (24), This is the reason of first instruction: addi sp,sp,-24



Scanned with CamScanner

Notes on leaf Procedure Example:-

① Push and Pop

بالأساس ال Stack معروف انها بتشتغل بالتخزين على مبدأ
→ last in first out

* لذلك من الطبيعي انك انت مجبورين لما نعمل restore from stack انه نكون نعكس

بعكس الترتيب ، الي ع لما نعمل ال save on stack ، ع اساس مش مجبورين

نعمل هيك ، مادام ال offset هو نفسه وال registers هي نفسهم

الاختلاف مرة وحدة.

لما نعمل ال local instructions بالترتيب ح ال SP shifting up

يعني بنعمل ال local لـ register واحد وبنعمل ال shift لل SP

وبعدين بنعمل ال local " الثاني ، و " " " SP وهكذا.

② Using X20 was not necessary.

We were able to save the value of (f) directly on (X10) which is the returning register that will hold value of (f) to return.

هيك كان عنا مختصر عملية ال [illegible] وال [illegible] وال [illegible]
الي علينا ال X20 ، وقلل عدد ال instructions

→ Register Usage for (leaf procedures)

→ Important notes in slide (57)

What if called the function:-
in the latest function of leaf procedure

example:-

for High-level

```
long long int main( )
{
  long long int z = leaf_example (5, 6, 7, 8)
}
long long int leaf_example(      )
  }
  {
```

x10 x11 x12 x13
e f g h

assembly code:-

```
addi x10, x0, 5
addi x11, x0, 6
addi x12, x0, 7
addi x13, x0, 8
jal x1, leaf_example
add x30, x10, x0 → returning value
```

x30 → for (z)

Scanned with CamScanner

Non-leaf procedure example:-

C code:

```
long long int fact (long long int n)
{
if (n<1) return 1;
else return n * fact (n-1);
}
```

How the code works:-

assuming we passed (3) to the function.

→ fact(3)

3<1 ? false 2

return 3 * fact(2)

2<1 ? false 1

return 2 * fact(1)

1<1 ? false 1

return 1 * fact(0)

0<1 ? true.

return 1

returns 6

Scanned with CamScanner

① Identifying registers to use:-

→ argument n in x10

→ result in x10 too

Analyzing code to write RISC-V code:-

① you need to store all of these - if exist

- saved registers
- return address
- maybe (x10) depending on the code
  ↳ maybe (x10)

→ for non-leaf procedure example (main-call)

→ [assumed]

PC	main:	
0	addi x10, x0, 2	
4	jal x1, fact	x1/r4 and 8(r4
8	addi x30, x10, 0	

Should take care about:-

① can determine what to save on stack in non-leaf procedure.

② what should happen after (jal)

Scanned with CamScanner

main call - non-leaf example:-

PC

0 addi x10, x0, 2

4 jal x1, fact

8 addi x30, x0, x10

argument = 2

Tracking

PC: 0 4 100 132 136 100 104 108 112
116 132 136 100 ---- 116 120 -- 128 140 --- 180
↓
140
⋮
180
↓
8

x1: 8 140 140 8

x10: 2 1 0 1 1 2

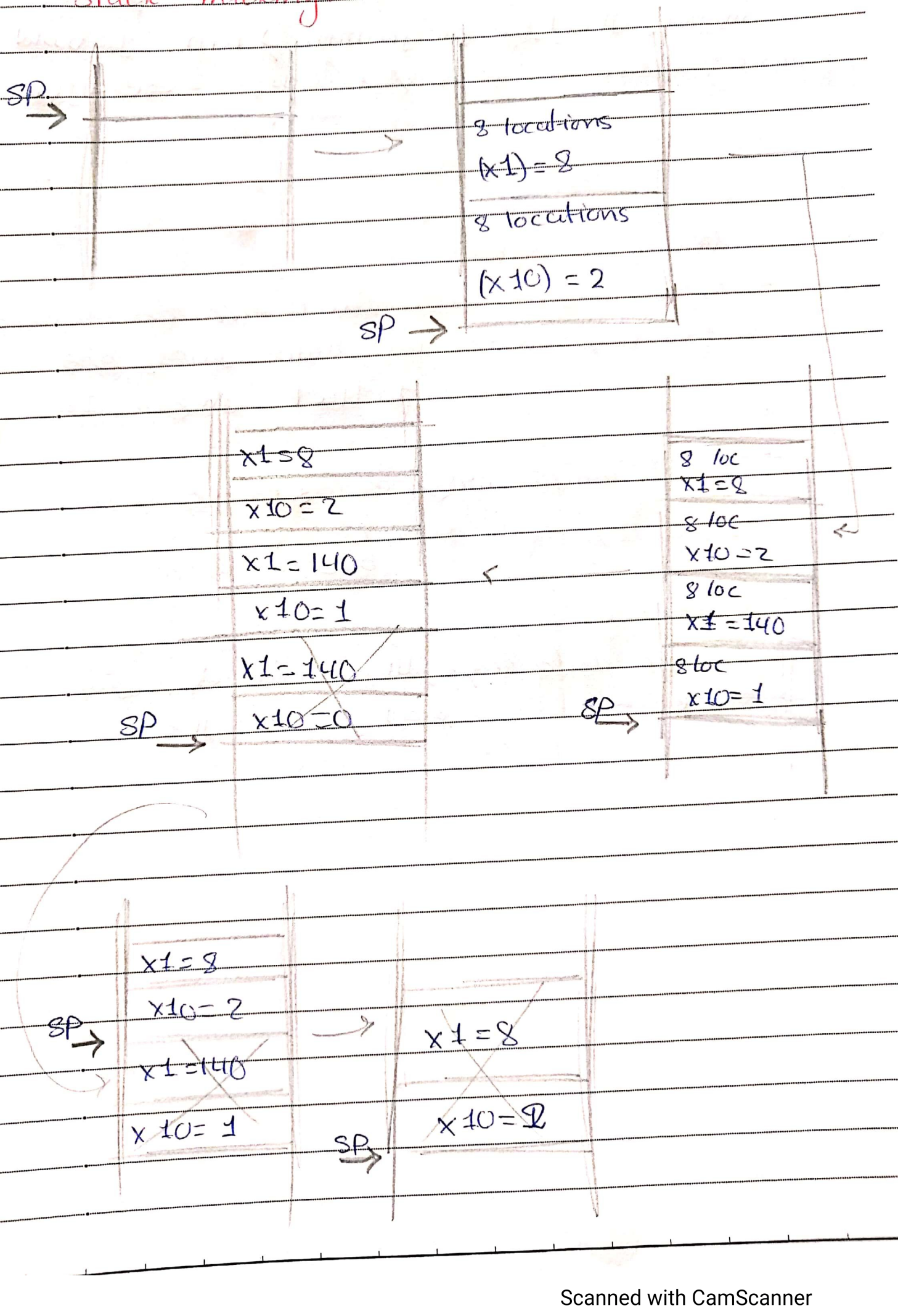
x6: 1 1

x30: 2

x5: 1 0 -1

Scanned with CamScanner

Stack tracing:



Scanned with CamScanner

لما بنشغل run ال programm بيكون ليه ال memory
هيا ليه كانه: virtual address space ← memory space.

Memory layout

→ static data :- static, fixed size for variables.
↳ global pointer : بيشاور ع static data
عشان نتحرك ± offset

offset limits: -2048 ≤ offset ≤ +2047
لـ for 12 bits available.

→ Dynamic data/heap بتبدأ من ال address الصغير للكبير
→ stack " " ال " الكبير للصغير.

Iteration Vs. Recursion

Iteration code:-

compile as while loop:-

→ compiler will se the code as leaf procedure for the next implementation-

```
long long int sum (long long int n, long long int acc
{
    while (n>0) {
    acc = acc+n;
    n = n-1; }
    return acc;
}
```

(RISC-V)

```
Sum:  bge x0, x10, Exit
      add x11, x11, x10
      addi x10, x10, -1
      beq x0, x0, Sum
Exit: add x12, x11, x0
      jalr x0, 0(x1)
```

→ leaf, no need to save return register
→ no need to use saved registers

Scanned with CamScanner

Byte / Halfword / Word Operations (for numbers data)

→ load byte 8-bit lb rd, off(rs1)
halfword 16-bit lh rd, off(rs1)
word 32-bit lw rd, off(rs1)
double word 64-bit ld rd, off(rs1)

offset: will be sign extended to 64 (12-bit)
to be able to add to the base
register sized with 64 bits

when loading 8, 16, 32, bits to the
registers, the values will be saved
at the side of least significant bits,
and rest of bytes are sign extended.

Scanned with CamScanner

String copy Example:-

creating char A[10]

null character '\0' → ASCII = 0

← يكون إجبارياً آخر char بال string

← هيك استخدمه لفصل ال strings عن بعضهم

→ This is called a null-terminated string ✓

data size for char x[ ], char y[ ]

one byte ✓

32-bit constants :-

wanna addi, sub, load a constant of bigger size than 12 bits to a register occasionally for 32-bit constant.

→ Think this way &

use ↓

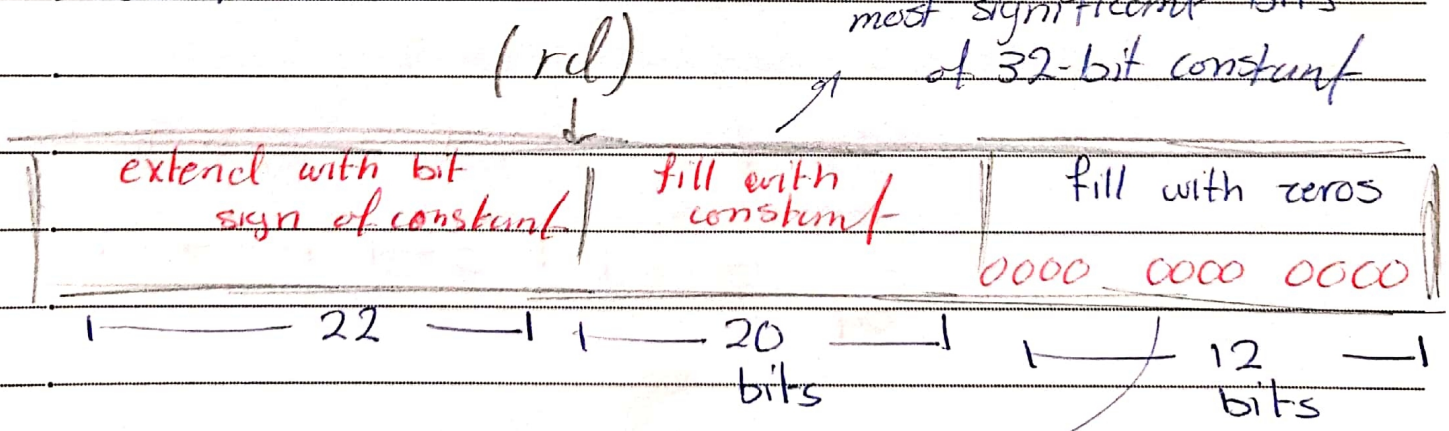
Load upper immediate

Lui rd, constant → of at most 20 bits

Then ↓

use addi to add the rest of constant bits to register (rd)

how it works :-



leave empty to fill it with the 12 least significant bit of constant by addi

Scanned with CamScanner

Two cases:

when 12-least sig bit are positive signed

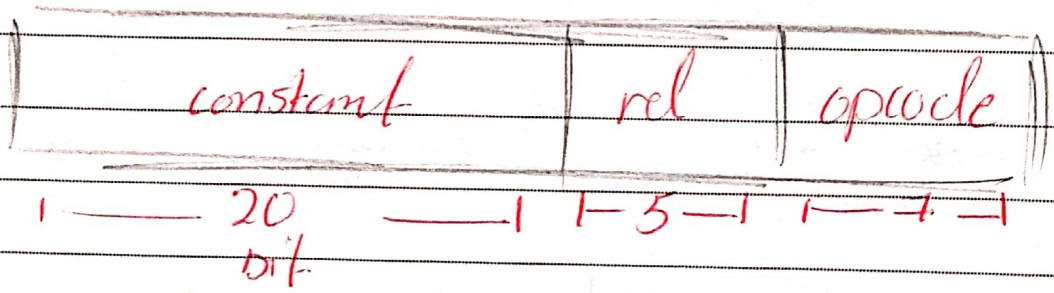
↓

implement it as usual

when 12-least sig bit are negative signed:

① at stage of first most 20 sig bit, add 1 to them, then complete implementation to the end as usual.

Format of lui (its own format):



Scanned with CamScanner

Branch addressing :-

Forward branch :- (immediate is (+))
target comes after the branch.

backward branch :- (" is (-))
" " before " "

immediate = offset in halfwords.

Two ways to understand that :-

① what happens internaly :-

beq x10, x20, loop → = imm
rs1 rs2

→ we'll get the target address by using imm.

target addr = PC + imm x 2

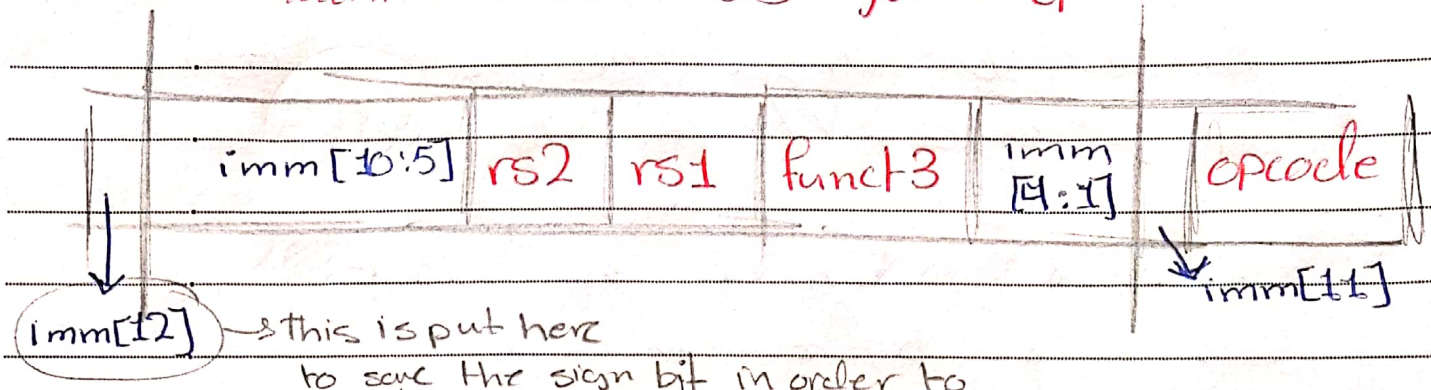
② what the immediate represent in fact :-

imm = offset in halfwords (how much of 2 bytes amount?)

from current PC to target PC

Scanned with CamScanner

How Branches are compiled to machine code:- SB format



→ this is put here to save the sign bit in order to do sign extension / always sign bit in the same place.

first before ~~else~~ compiling the imm, multiply it by 2, by shifting left by 1 bit, for the equation

target = PC + imm x 2

Then this goes ↓

a constant of 13 bits

→ we'll put in the format only 12 bits [12:1]

Scanned with CamScanner

Unconditional jump-and-link addressing

same as SB and branches principles

immediate unit = halfwords (2 bytes)
2x " " = word (4 bytes)

when transferring into machine code:-
if the imm or offset is negative.
find 2's complement of it, then, assign it to its place.

long jumps and branching for a way.

① lui first
② jalr second

lui x6, 0x[20B1
jalr x0, 0x004(x6)

If sign bit of offset is 0, implement it as usual

if sign bit = 1

Scanned with CamScanner

Some steps :-

But do the next:

go to address 0X 0000 0000 D321 A800

lui x5, 0XD321 (B) → not A because 800 is 1 signed.

addi x5, x5, 0X 800 ① → create the shift (pure) in x5

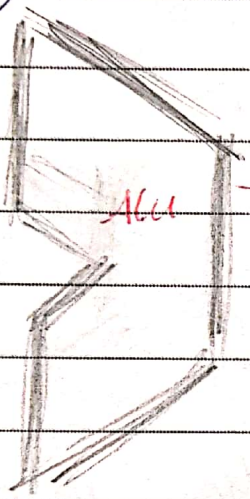
addi x6, x0, 0XFFF
srli x6, x6, 32
and x5, x6, x5 } ② → create the operand destiny with (and)

jalr x0, 0(x5) ③ → jump to address

Scanned with CamScanner

Appendix A.5
constructing a Basic Arithmetic logic unit (ALU)

→ Range of signed numbers in 2's complement format is imbalanced between positive and negative numbers.



→ ALU sign

Alu → Three inputs (A, B, m)

→ Three outputs (zero, ov, result)

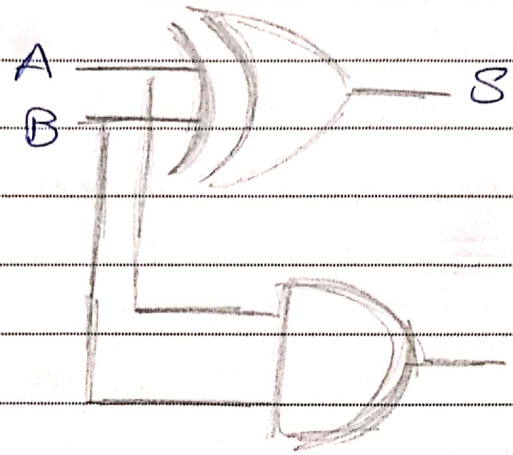
any handling for extensions is done out of ALU:- sign extension, unsigned-extension

When designing ALU: generate 1-bit ALU slice, the replicate 64 times

(using, And, OR, inverters MUXes

Scanned with CamScanner

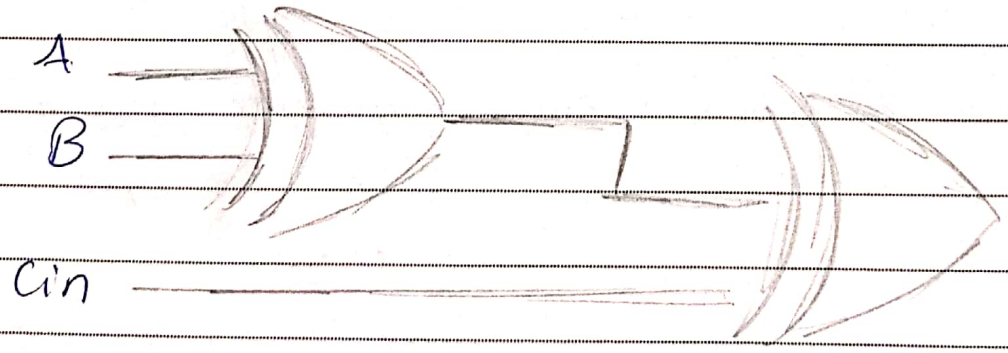
Half (2,2) Adder

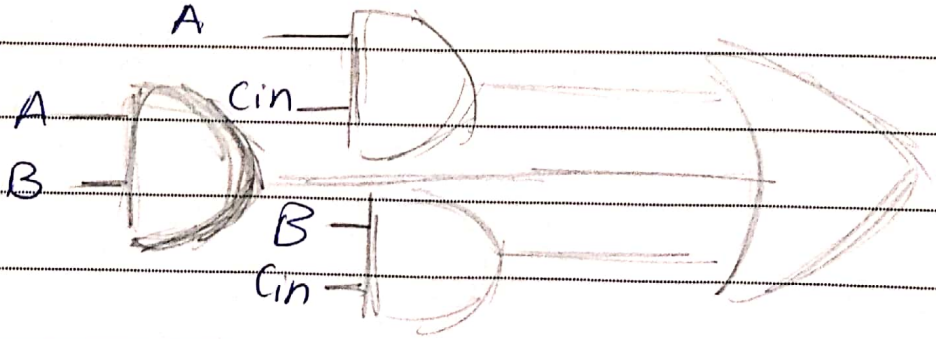


Full (3,2) Adder

writing odd parity function in other way:

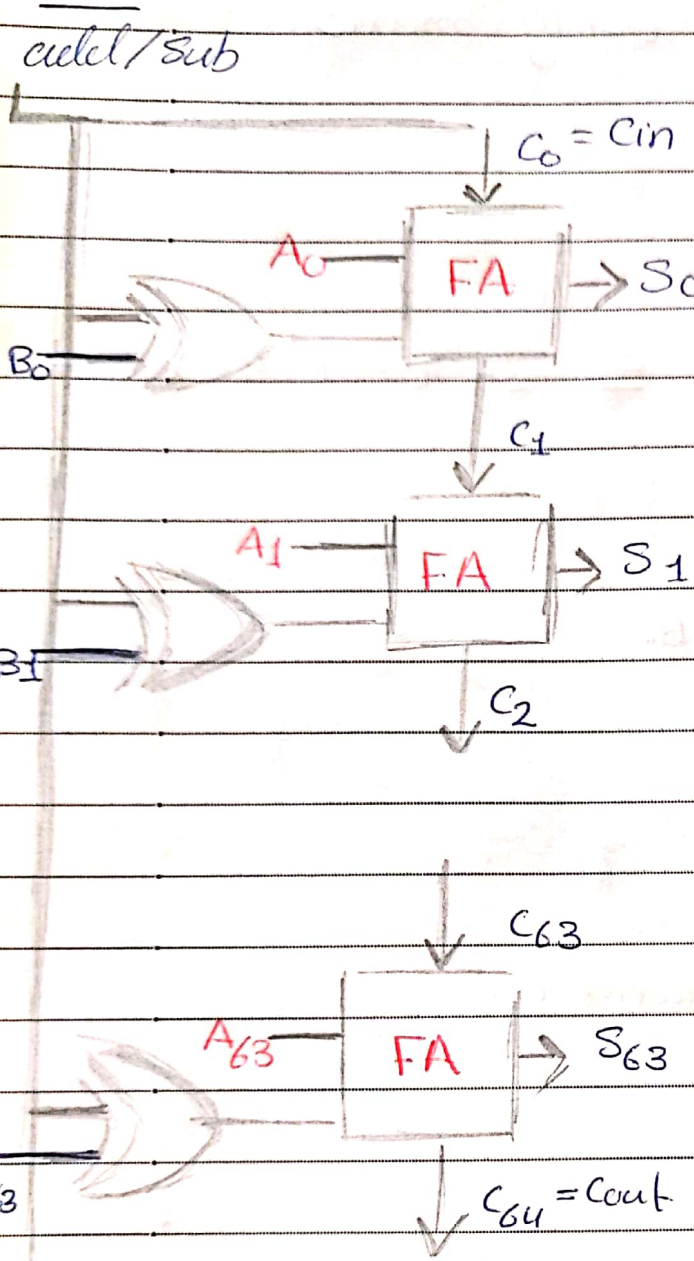
$S = \bar{A}.\bar{B}.Cin \mid \bar{A}.B.\bar{Cin} \mid A.\bar{B}.\bar{Cin} \mid A.B.Cin.$





Scanned with CamScanner

The 64-bit Ripple carry adder/Subtractor



when implementing addition:-

$C_{in} = 0$

A+B is implemented.

→ XOR with zero:-

$0 \oplus X = X$

$0 \oplus 0 = 0$

$0 \oplus 1 = 1$

when implementing subtraction:-

$C_{in} = 1$

$A - B = A + \bar{B} + 1$

→ $\bar{B}$ will be found by xor gate

$1 \oplus X = \bar{X}$

$1 \oplus 0 = 1$

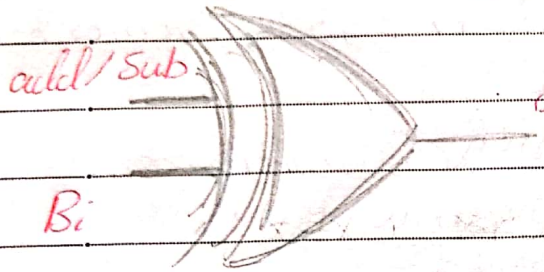
$1 \oplus 1 = 0$

→ 1 will be added through the cin input

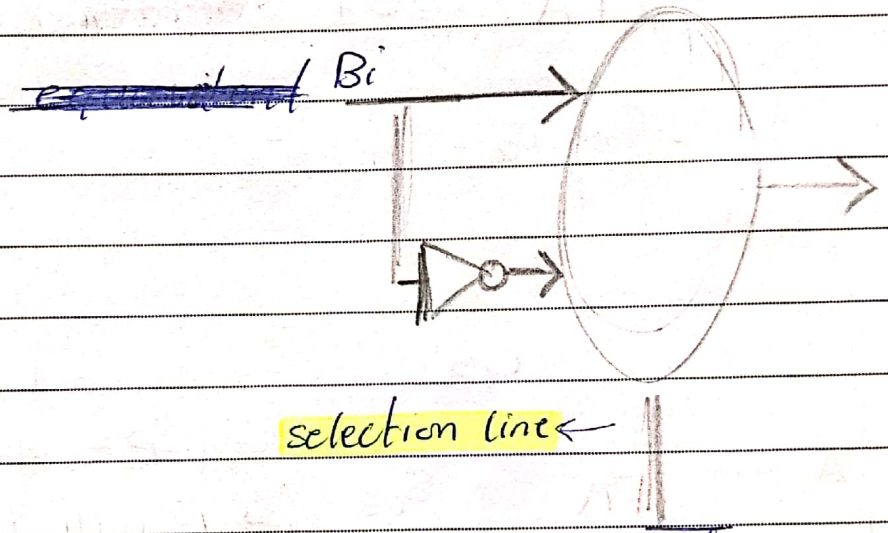
Scanned with CamScanner

The other design for the Adder / Subtractor using muxes

original design



$\equiv$ 2-to-1 MUX

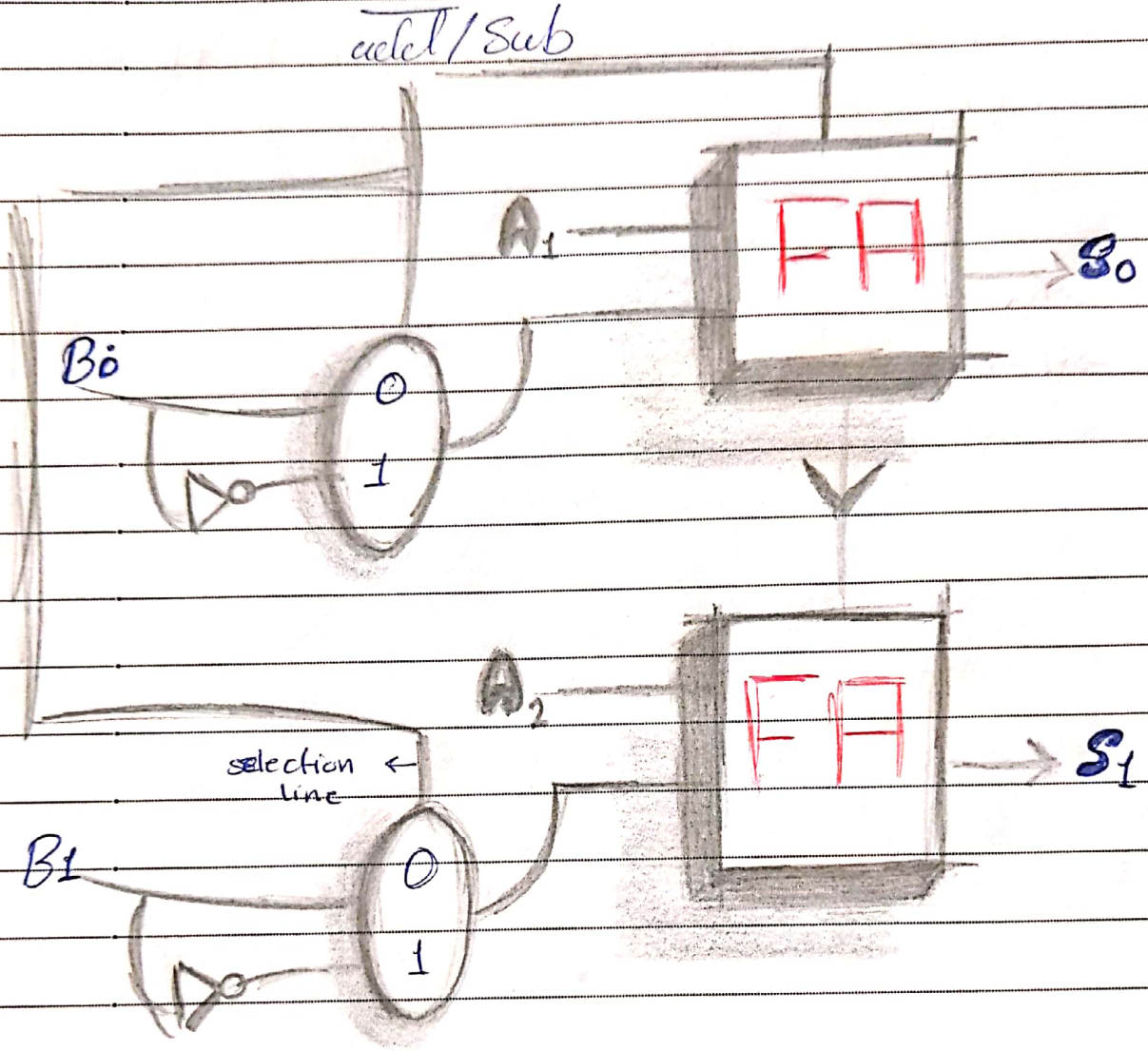


selection line ←

add / sub = 0
or
= 1

Scanned with CamScanner

Example: 2- bit adder/Subtractor using MUXes



XOR بدل ال MUX كمان B بدل كل

Scanned with CamScanner

→ Ainvert, Binvert

① when having value 0, use a_i and b_i as are

② when having value 1, invert both a_i and b_i

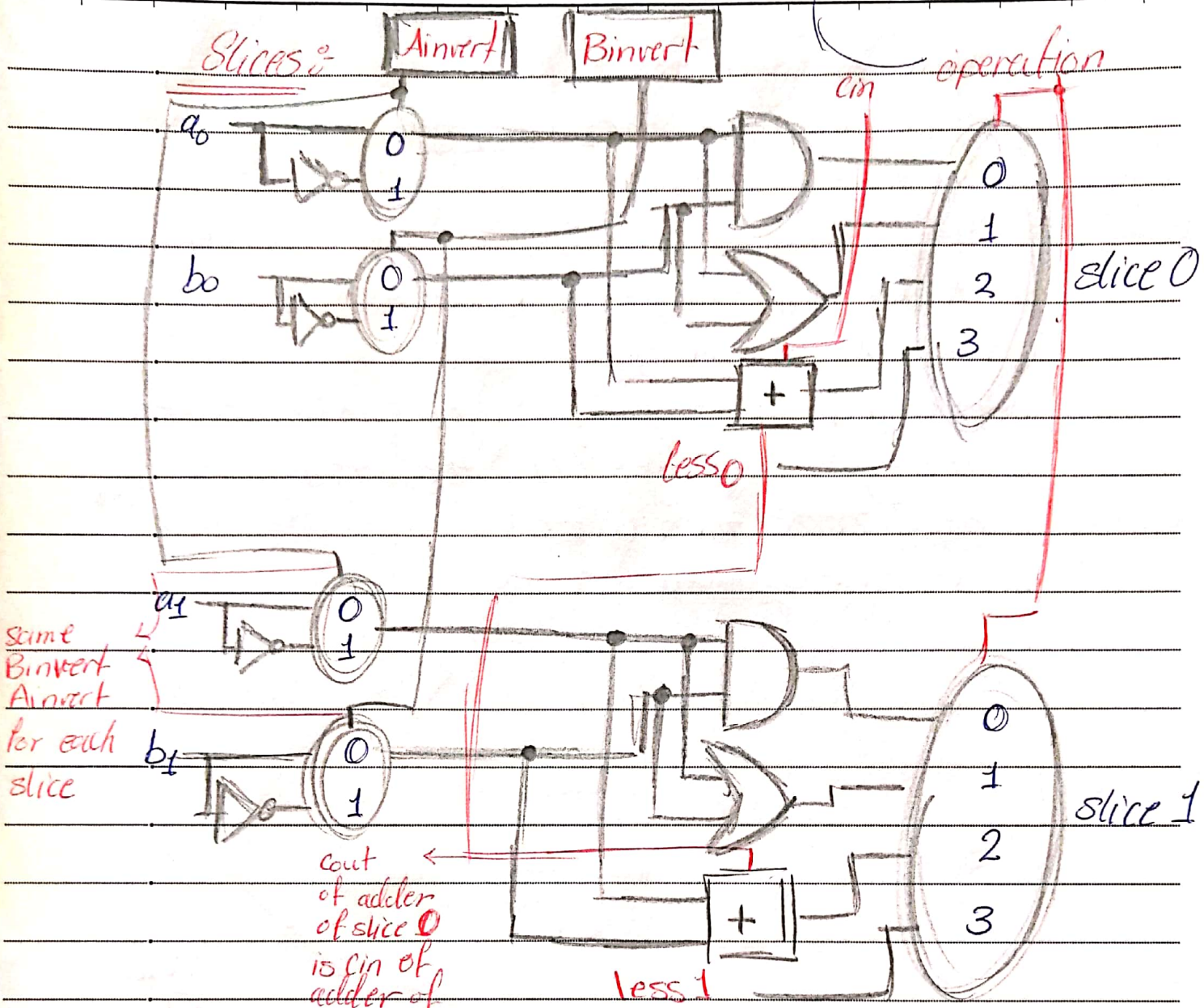
→ Mux of operation

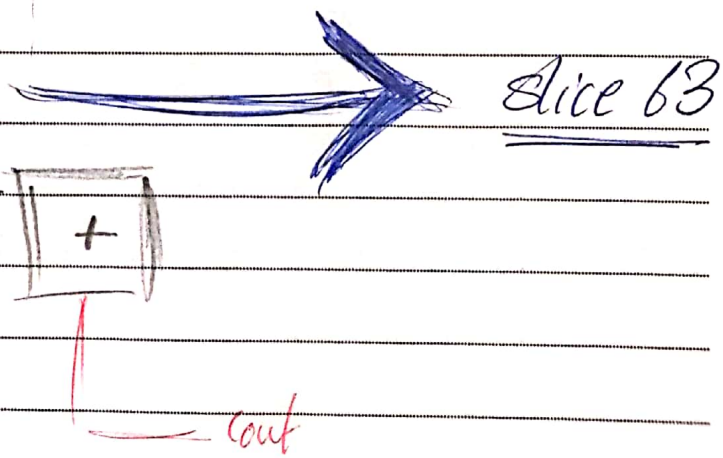
→ 4-to-1 Mux

→ 4 inputs, 1 output

Scanned with CamScanner

same operation selection line of 2 bit for each slice.





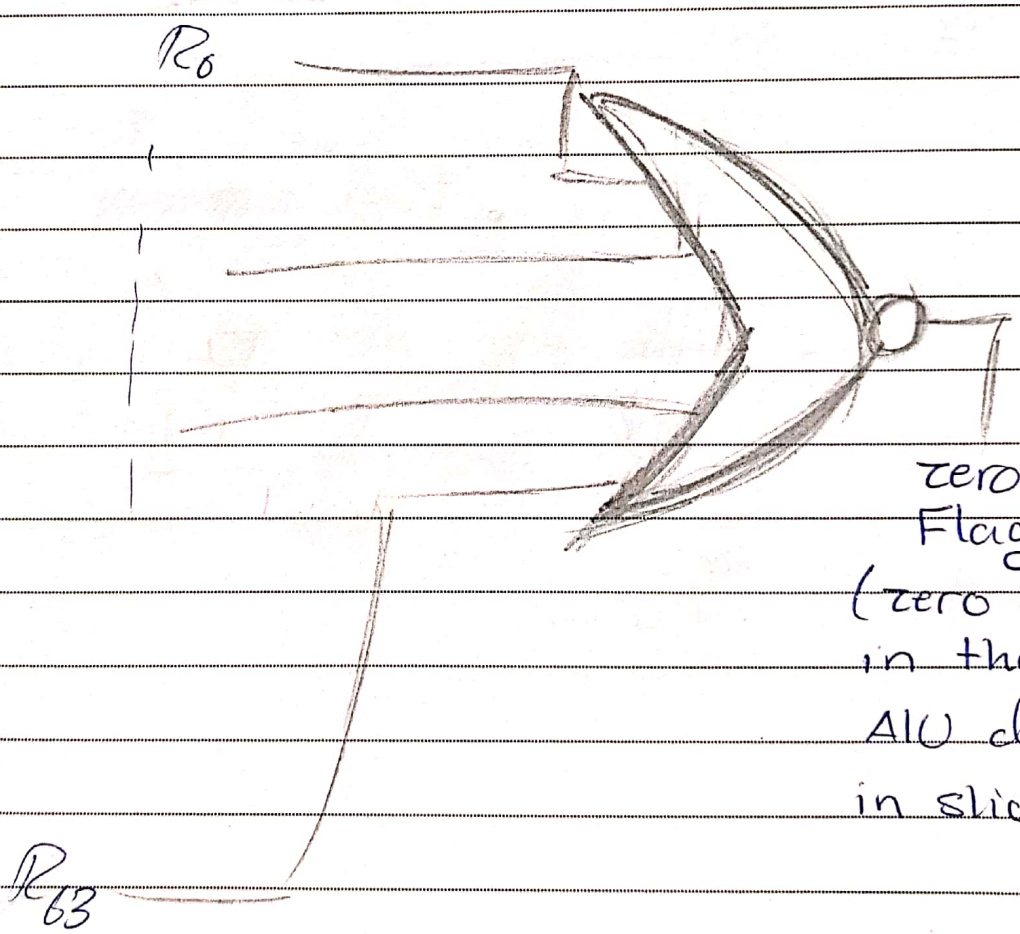
Scanned with CamScanner

How to implement (bne, beq)

→ use (a-b) the compare result with (0)

to know if result is 0 or not

- use NOR gate.
  - connect R_0 → R_{63} with Nor gate
  - if only 1 bit was (1), the result is not zero, then (a≠b)



Scanned with CamScanner

Set if less than (SLT) implementation:-

① How the final result is performed

→ note that all bits of this instruction result are going to be (0) except the least significant bit, which may be (1), and is (R_0)

→ Hence:- all bits (all results are set to ground starting from R_1 ending up with R_{63})

→ R_0 is left free to be set by the hardware element while implementing instruction.

② But how internally is that done?

if instruction is slt rd, rs1, rs2

→ Alu uses the adder/subtractor in its design to implement (rs1 - rs2)

→ Check the result:-

* if negative (bit Alu_{63} is 1), then rs1 is really less than rs2
* if zero/positive or zero (Alu_{63} is 0), then rs1 is not less than rs2.

Hence:- Alu itself is used for the set (less 0) the wire of [63] result of adder/sub (63) is connected with less 0 to set it to the wanted value.

Scanned with CamScanner

كل ال ← Alu design في ال slices مستخدمين

Alu_{63} و Alu_0 ما عدا

→ Alu_0 : least significant Alu slice:-

① has an external input ($cin \equiv C_0$)

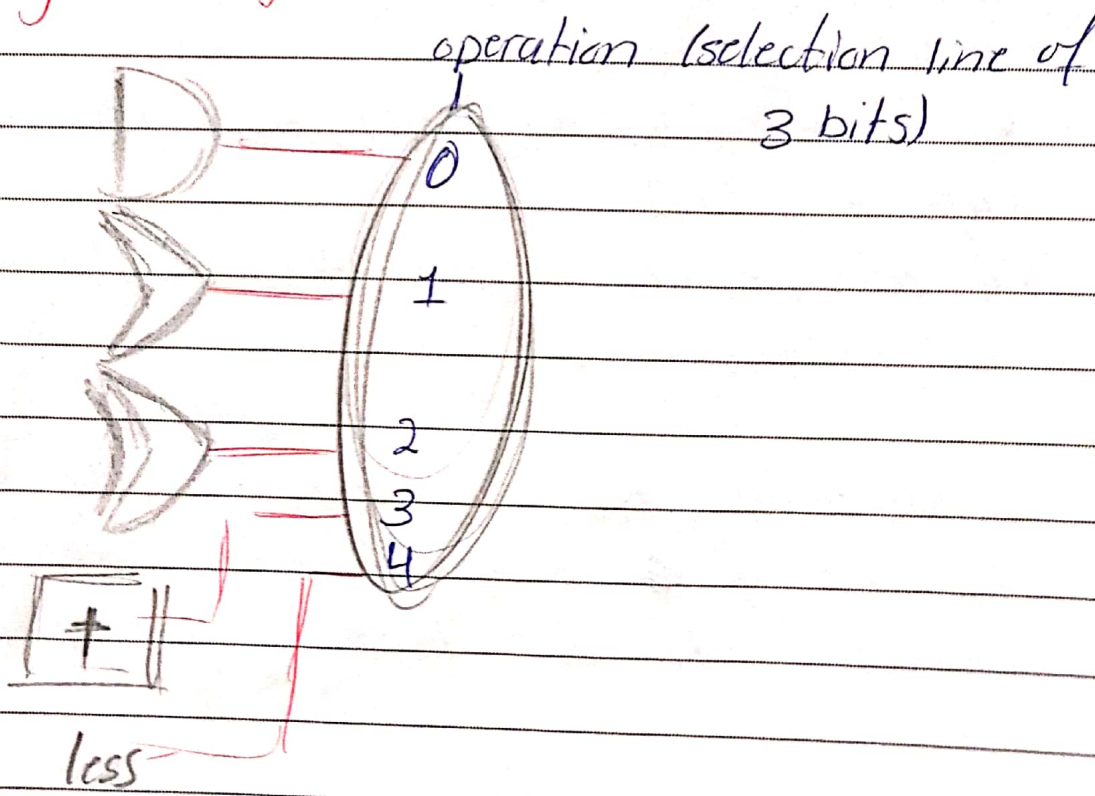
② its ($less \equiv less0$) is connected to add/Sub result of Alu_{63}

→ Alu_{63} : Most significant Alu slice:-

① has two additional outputs:

- cout of its add/sub
- output of add/sub which is connected to $less_0$ which is called (set)

if you want to desig Alu ~~steel~~ slices using XOR gate too:



Scanned with CamScanner

To understand NOR operation:

→ remember:

For all logical operations (Except NOR):
Binvert = 0, and $C_0 = x$

→ NOR: Binvert = 1, $C_0 = x$

↳ NOR ~~add~~: nor rd, rs1, rs2

↳ (rd) = not ((rs1) OR (rs2))

↳ apply demorgun's law ↓

↳ (rd) = (not (rs1)) AND (Not (rs2))

⇒ So NOR can be done without adding any new element to the ALU, just by implementing AND to inverted values of rs1, rs2 ✓

Then Ainvert = 1
Binvert = 1
operation = 00 (AND)
$C_0 = x$ (don't matter)

Scanned with CamScanner

[Appendix A-6]

check delay :-

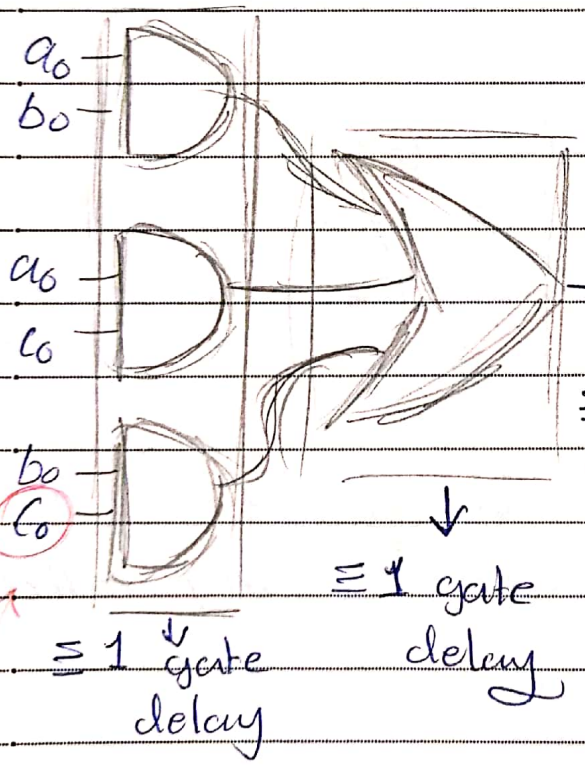
→ Ripple - Carry Adder / RCA

→ evaluate delay for Cout , and S

Cout → carry out.

S → sum

Cout

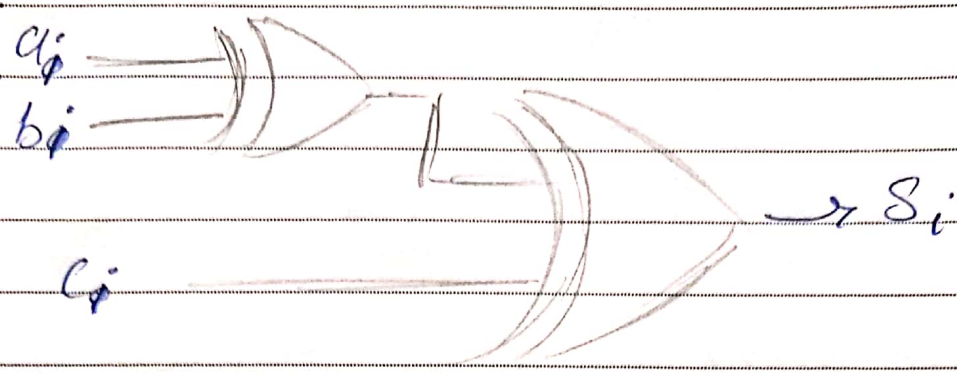


S

Then for each one on the line you added 2 to it, because of previous Cout

Scanned with CamScanner

Sum delay:



Example:-

① S_0

a_0, b_0 are ready at delay = 0

c_0 is ready at delay = 0

XOR ≡ complex gate.

XOR delay = 2 gate delay

so in total S_0 delay = 4 gate delay.

② S_1

a_1, b_1 are ready at delay = 0

→ c_1 is ready at delay = 2 in parallel with XOR of a_1, b_1 of delay = 2

→ second XOR = 2 delay

Total delay = 4

Scanned with CamScanner

(3) S_2

a_2, b_2 are ready at delay = 0

→ c_2 ready at delay = 4 // XOR2 = 2 delay

= 4 delay.

→ second XOR2 = 2 delay

Total = 6

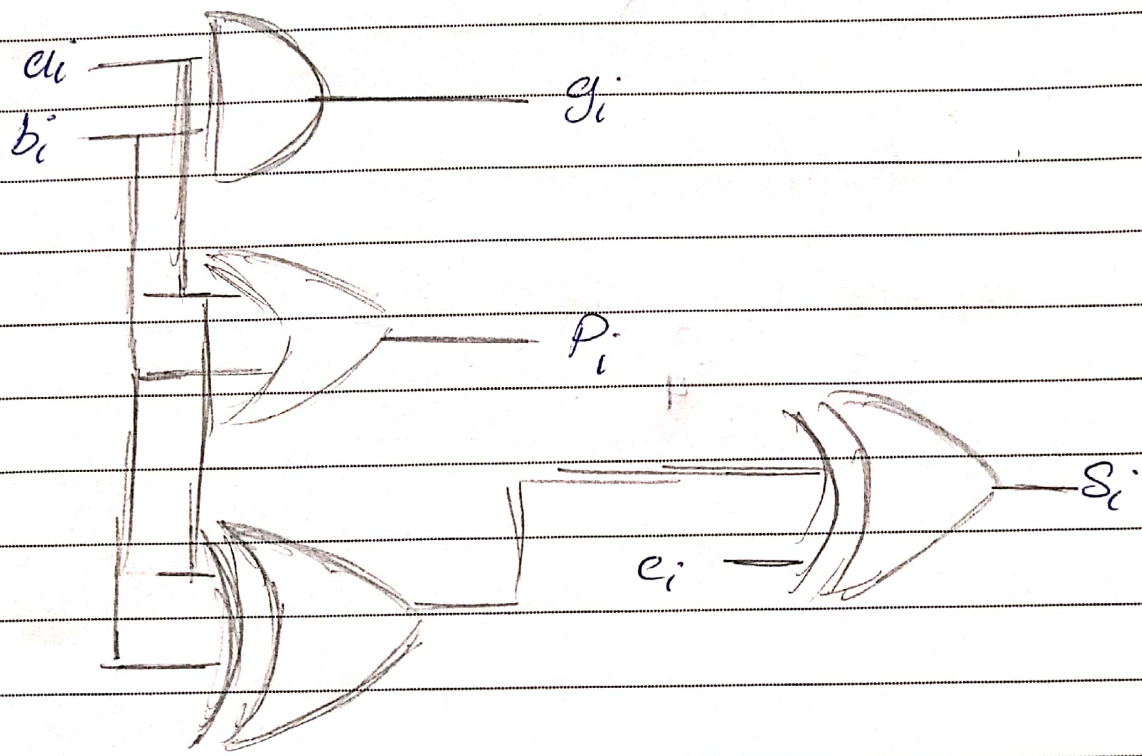
Hence : S_3 delay = 8

C_4 // = 8

Scanned with CamScanner

Carry-Lookahead Adder (CLA)

new design of FA

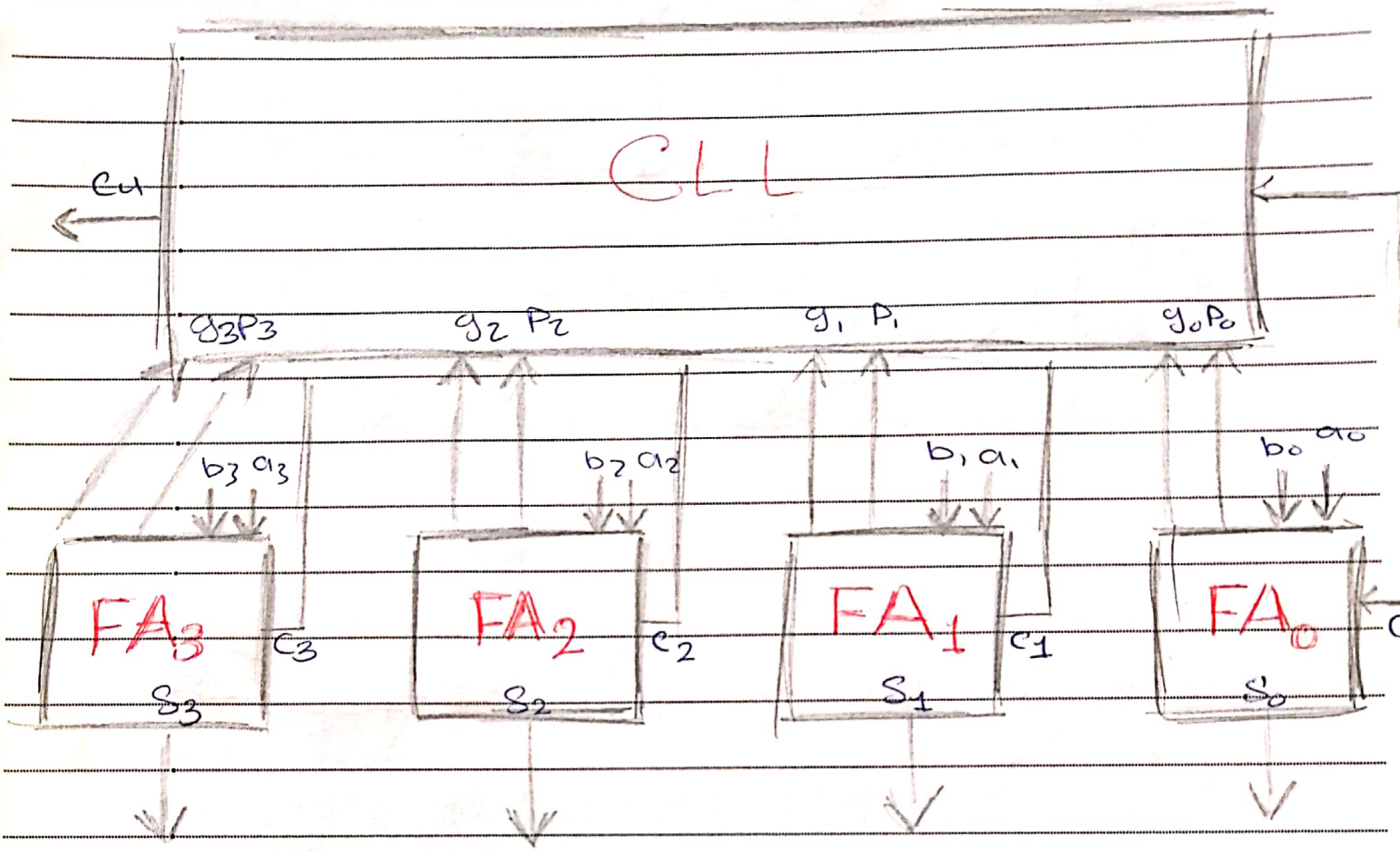


what is g_i (generate)

→ to find Cout without knowing C_{in} necessarily [يعني ال FA حطله carry ولا لأ]

what is p_i (propagate)

→ to hold the carry from input to output.



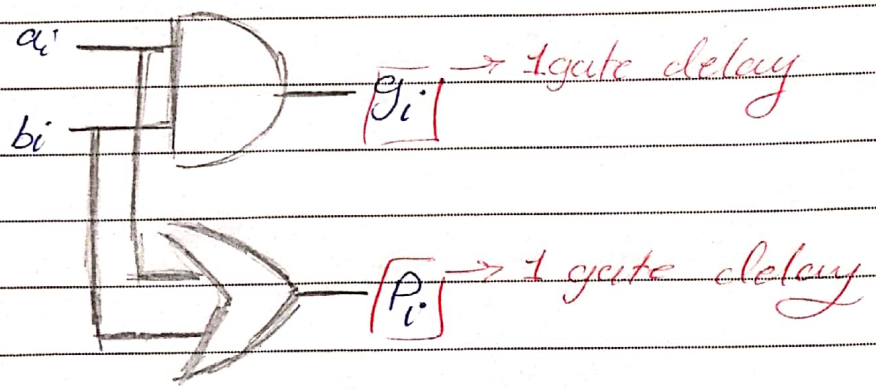
Carry Equations:-

C_1 C_2 C_3 C_4

depend on propagate, and generate

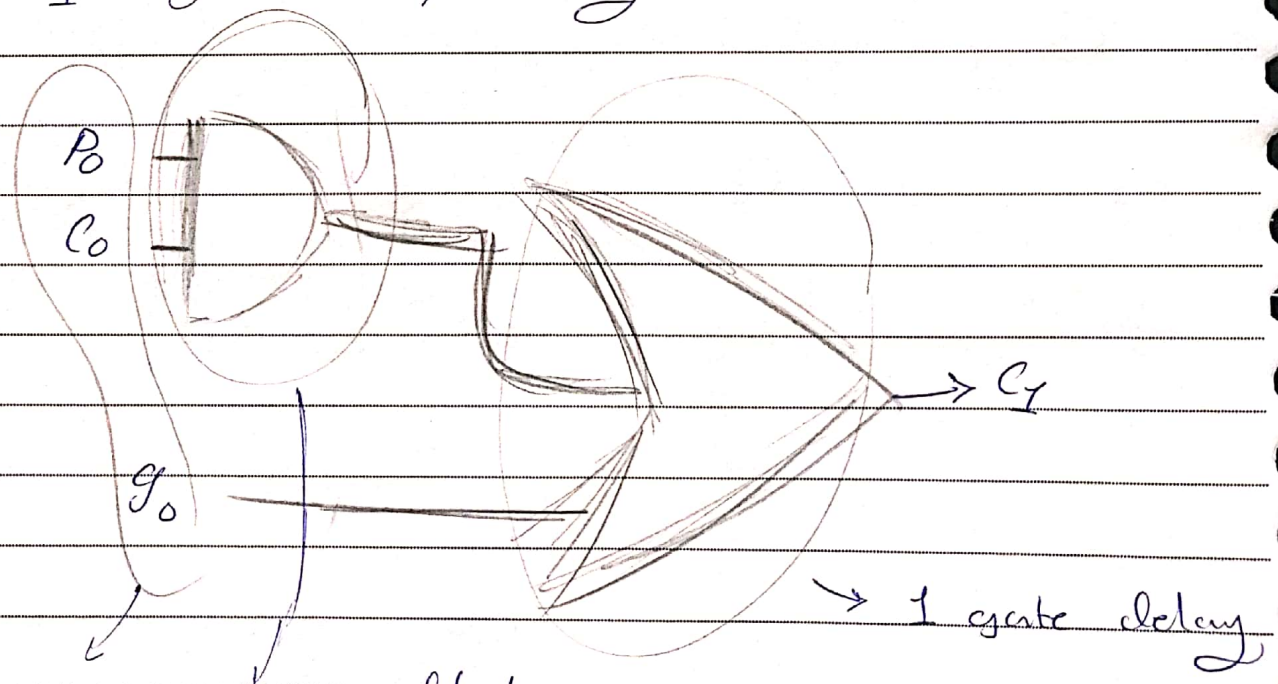
Scanned with CamScanner

Delay of g_i and p_i :



So for each carry you'll need ~~4~~ it'll take 3 gates delay, ~~to 3 gate~~ 1 of them goes for p_i and g_i in parallel.

C_1 logic / depending on its equation.



1 gate delay

1 gate delay

3 gates delay in total

and so on

Scanned with CamScanner

when FA_3 generates nothing, and propagates nothing

$g_3 = 0$
$p_3 = 0$ $\rightarrow$ $c_4 = 0$

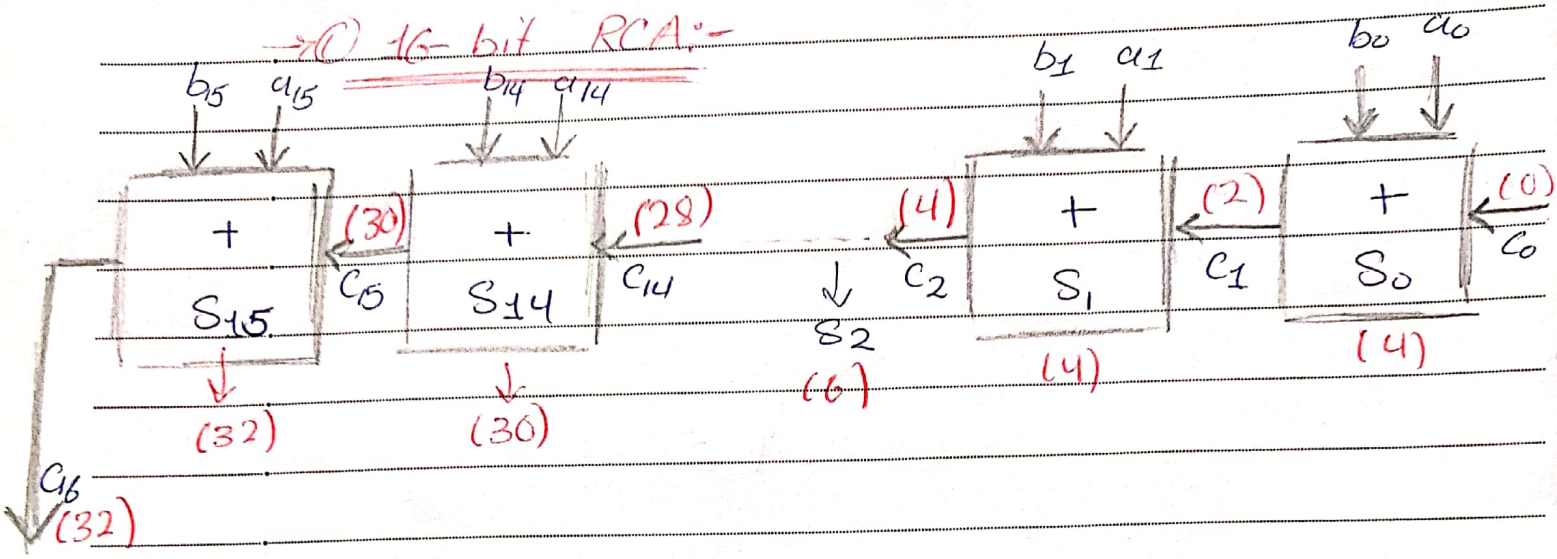
→ Delay of 4-bit CLA = 5 gate delay

→ Delay of 4-bit RCA = 8 gate delay

Scanned with CamScanner

16-bit designs:-

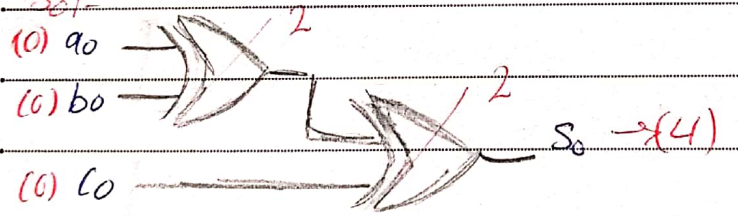
(1) 16-bit RCA:-



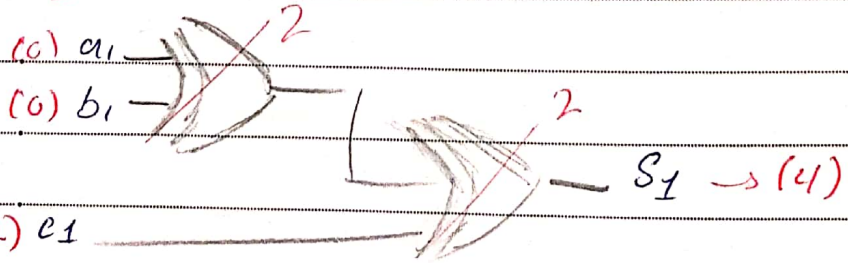
Delay of 16-bit RCA = 16 x 2 = 32 gate delay.

for S delays:

S_0 :-



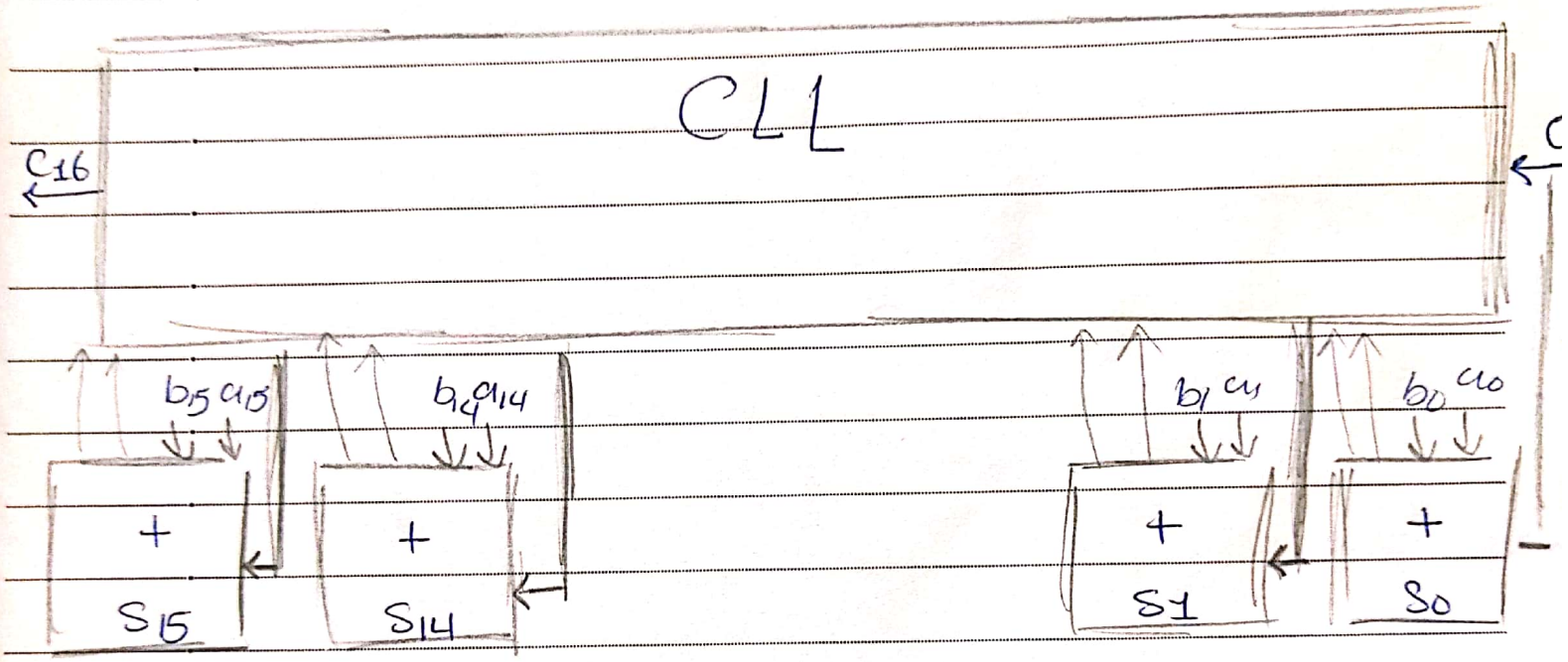
S_1 :-



Scanned with CamScanner

* delay of the carry = 2x its order
* delay of the largest S = largest C delay

(2) 16-bit CLA design (# 1)



Equation of C_{16} =

$$g_{15} + P_{15} g_{14} + \cdots\cdots + \cdots$$

$$\cdots + P_{15}P_{14}P_{13}P_{12}P_{11}P_{10}P_9P_8P_7P_6P_5P_4P_3P_2P_1P_0C_0$$

17 inputs for one AND Gate // not practical

17 " " " OR "

Scanned with CamScanner

Chapter 4 // Processor

§4.1 // Introduction:

Instruction Excution:-

[1] fetch :- get the Instruction from memory.
- Instruction whose address is saved in the register PC/program counter.
- Instructions are 32-bit length

[2] read registers : - get their numbers from the instruction you've got by (fetch)
- go to register file and read those registers.

[4] Access // only for ld/sd
→ read or write

[5] * write on destinating register (rd)
↳ for R-type and ld instructions.

* change PC to PC+4 to move on
OR
* change PC to target address (IF needed)

Scanned with CamScanner

§ 4.2 // logic design basics

Edge-triggered

+ve : update when Clk (0 → 1)

-ve : " " " (1 → 0)

Clocking Methodology

- The old value is the value during the clock cycle.

Scanned with CamScanner

Control unit is in slide 6

→ CPU Overview:-

* Control signals:- (slide 6)

[1] all used multiplixers are: 2-1 muxes
- two inputs
- [one-bit] selection line

[2] the input of the control unit: Important

The Opcode coming from the instruction bits: instru[0:7]

[3] Control signals out the control:-
[A] Alu operation (R-type instructions case)
[B] MemWrite / MemRead (Data memory)
and they both maybe 0,0.

[C] Branch , Reg Write.

Scanned with CamScanner

Logic Design basics:-

Two types of elements in design:-

[1]	[2]
Combinational	Sequential
- And	
- Adder	
- Mux	
- ALU	
output is a function of input	

→ [Sequential Elements] :-

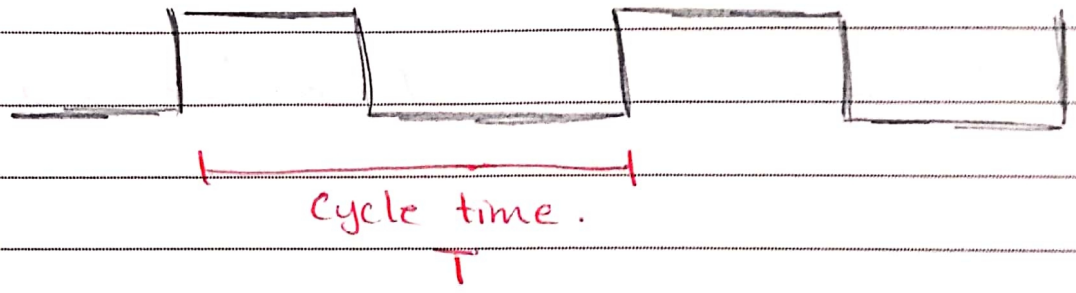
Remember:-

① Edge-Triggered :-
+ve → upgrade when clk (0→1)
-ve → " " clk (1→0)

② What is register:- (1-bit register)
it is a FlipFlop.

Scanned with CamScanner

③ A signal (clk)



Then clk frequency = $\frac{1}{T}$

Important:- Register with write control:- only updates on clock edge when write control input = 1

Scanned with CamScanner

Topic :- Clocking Methodology :-

→ Why is it important to define when to read or write?

to read → @ (current / old value)
to write → @ (when write = 1, +ve edge)

→ to determine and have complete understanding, about when the read value is to be, the old, or new value, or a mix

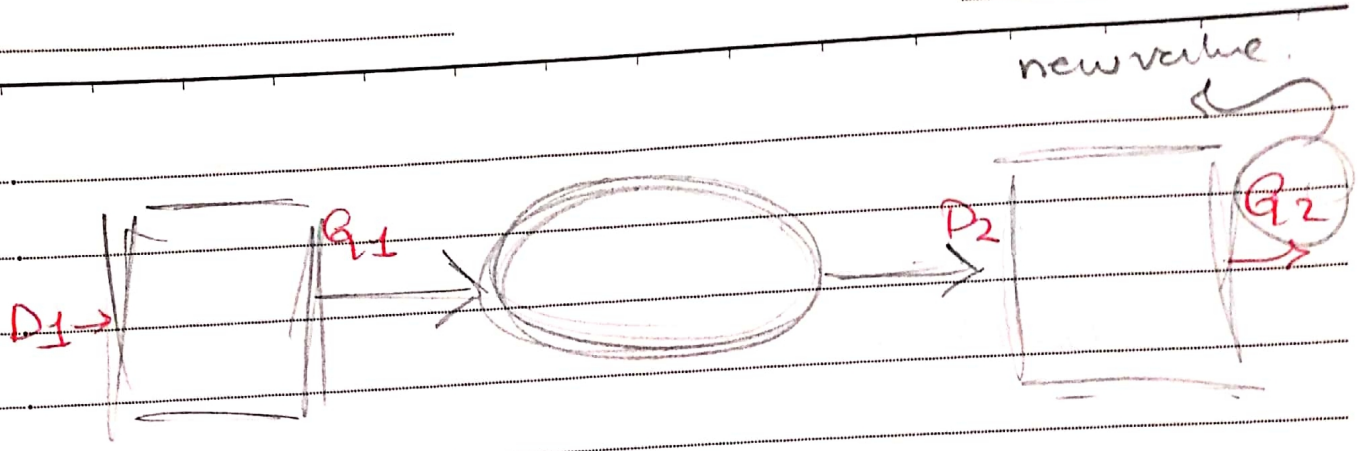
✱ Edge-triggered clocking methodology :-

[1] @ value (old) is always available during the cycle, you can read it any time.
from the begining of clock cycle, till the end

[2] During this clock cycle, you can do any operation on your (@) / Combinational operations ✱

Note that :- combinational logic should be finished before the end of the cycle, to be able to write the resulting value.

Scanned with CamScanner



Process :-

During the clock cycle:
- implementing the combinatio operations
- Q_1 (old value) can be read any time through clock cycle.
→ old value : value that the operations are done on.

Positive Edges :-
- contents of states can be read on (at beginning)
- writing the outputs of the combinational operations to the same state element or another.

Scanned with CamScanner

First Design Single-Cycle CPU

→ Building Datapath

① any element needed more than once must be duplicated (No datapath resource can be used more than once per instruction)

While tracking, you'll be tracking only for one instruction

Scanned with CamScanner

[1] Instruction Fetch:-

→ PC address is 64-bit

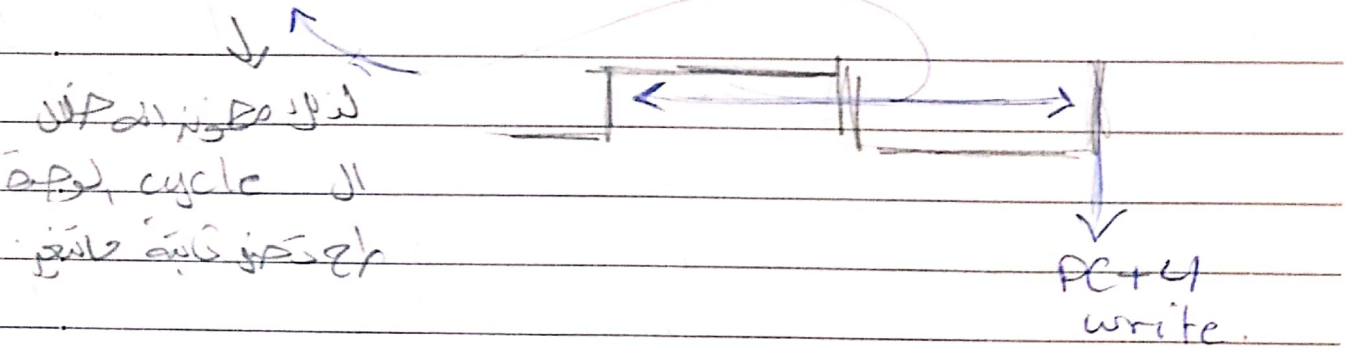
→ Instruction [0:31]

→ A part of Fetch is to prepare (PC+4)

PC register:-

→ only changes its value on positive edge (at the end of cycle)

→ through the cycle PC is staying the same.



Scanned with CamScanner

[2] R-Format Instructions

→ Two Registers operand (read)
(rs1), (rs2)
happening in Register file
take 5 bits from Instr[31:0] for each register to know which register to read.

reading
- → reach (rs1) → read ~~upload~~ get contents ~~value~~ to [Read data 1] 64 → bit
- happens in parallel!
- → reach (rs1) → read ~~upload~~ get contents to [Read data 2] 64 bits

Read Port:- مكان بنحط فيه address وهو بيطلع داتا
- → (rs1) → Read data 1
- → (rs2) → Read data 2

* we have two write ports in RF

writing
- → determine the destination register by Instr[31:0] by taking 5 bits from it in (write register) (reach)
- → Write the result on [Write data]

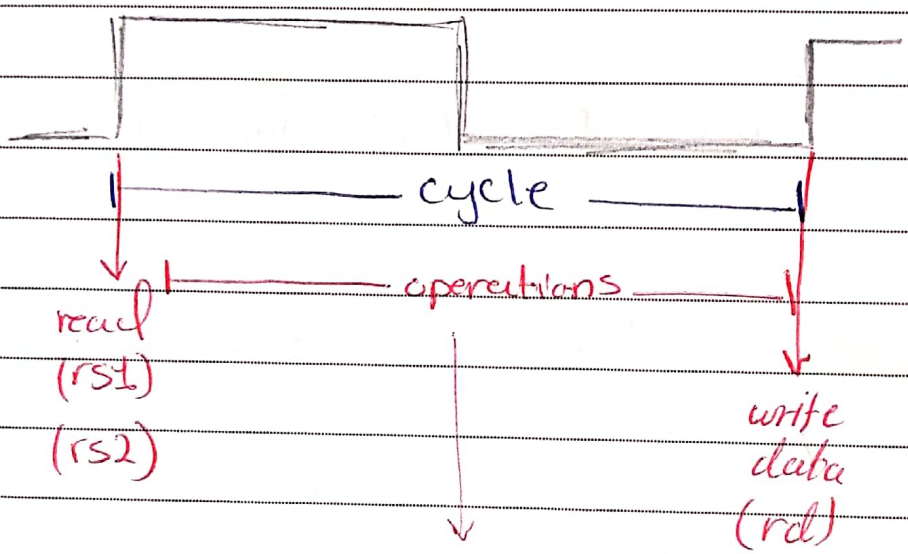
Scanned with CamScanner

Write Port:-

↳ (write register) → (Write Port)

5-bits 64-bits

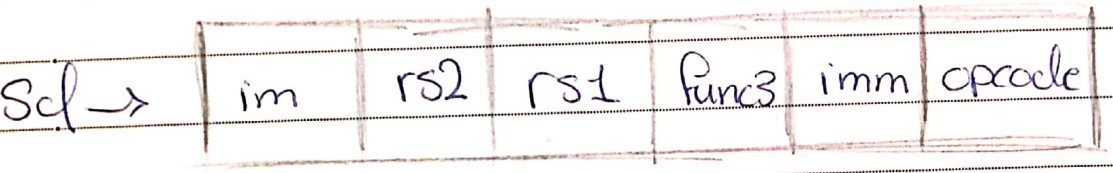
[Clocking methodology]



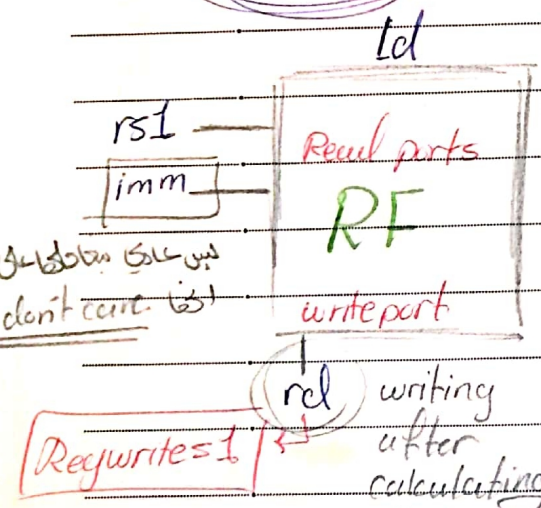
Scanned with CamScanner

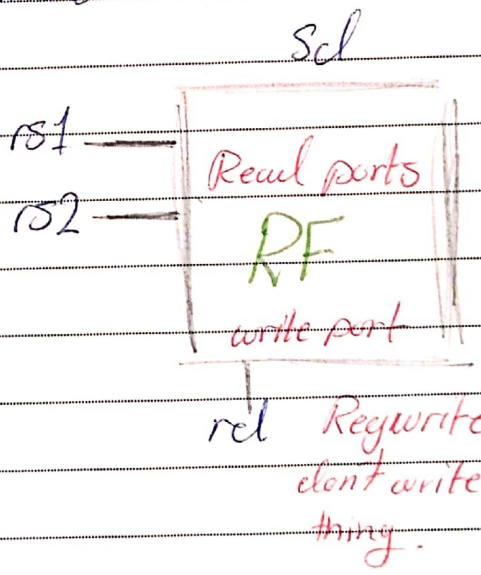
[3] load / Store Instructions

first.



Stage 1 read register operands.





Scanned with CamScanner

Stage 2 calculate address using 12-bit offset.

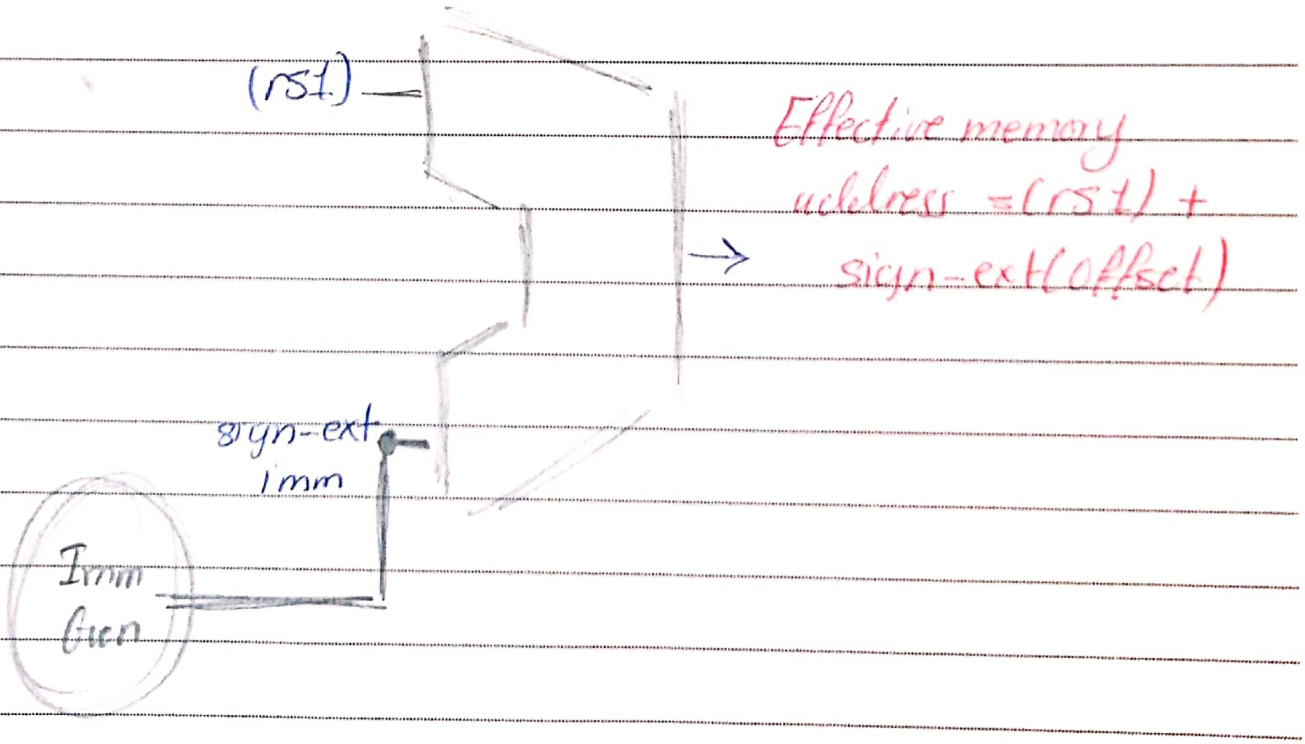
Sign-extension in Immediat Generator → ALU calculations.

[1] Imm Gen

Input: Instruction [31:0] → Imm Gen → 64-sign-ext-imm.

For both load, store.

[2] ALU calculations



Scanned with CamScanner

Stage 3 Handling to Data Memory

[1] load: Read memory and update register (rd)

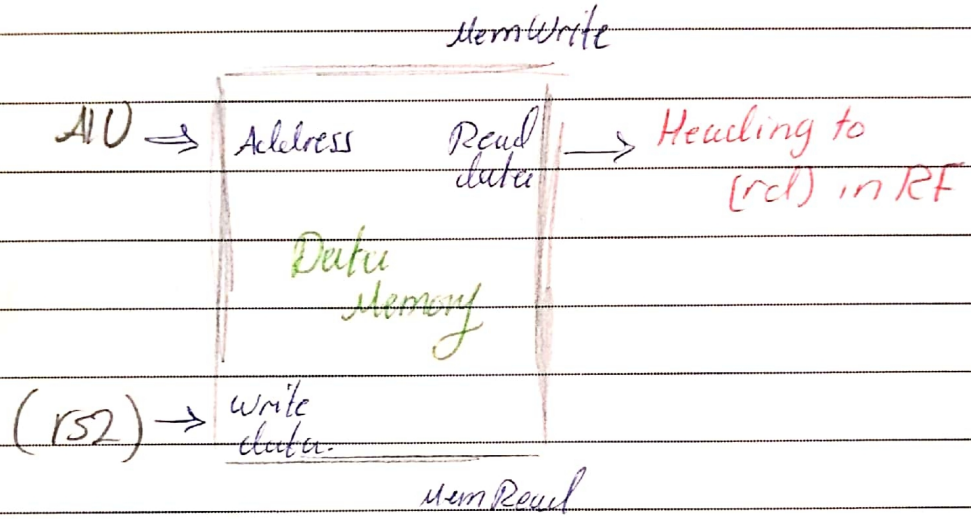
→ MemRead = 1

→ MemWrite = 0

[2] Store: Write register (rs2) value to memory

→ MemRead = 0

→ MemWrite = 1



Scanned with CamScanner

[4] Branch Instructions:

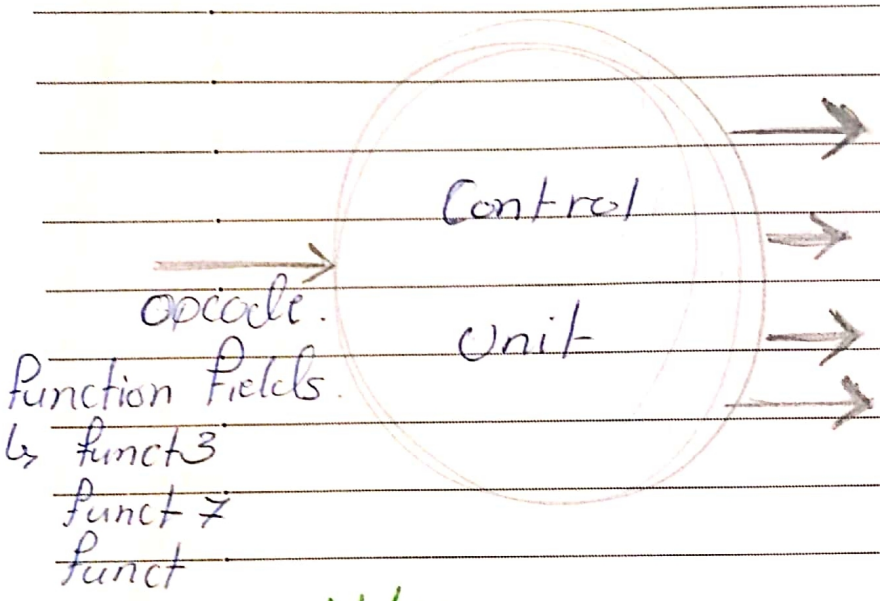
BTA = Branch Target Address.

- $BTA = PC + \left(\begin{matrix} sign \\ -ext \\ imm \end{matrix}\right) \times 2$

shift left 1

Scanned with CamScanner

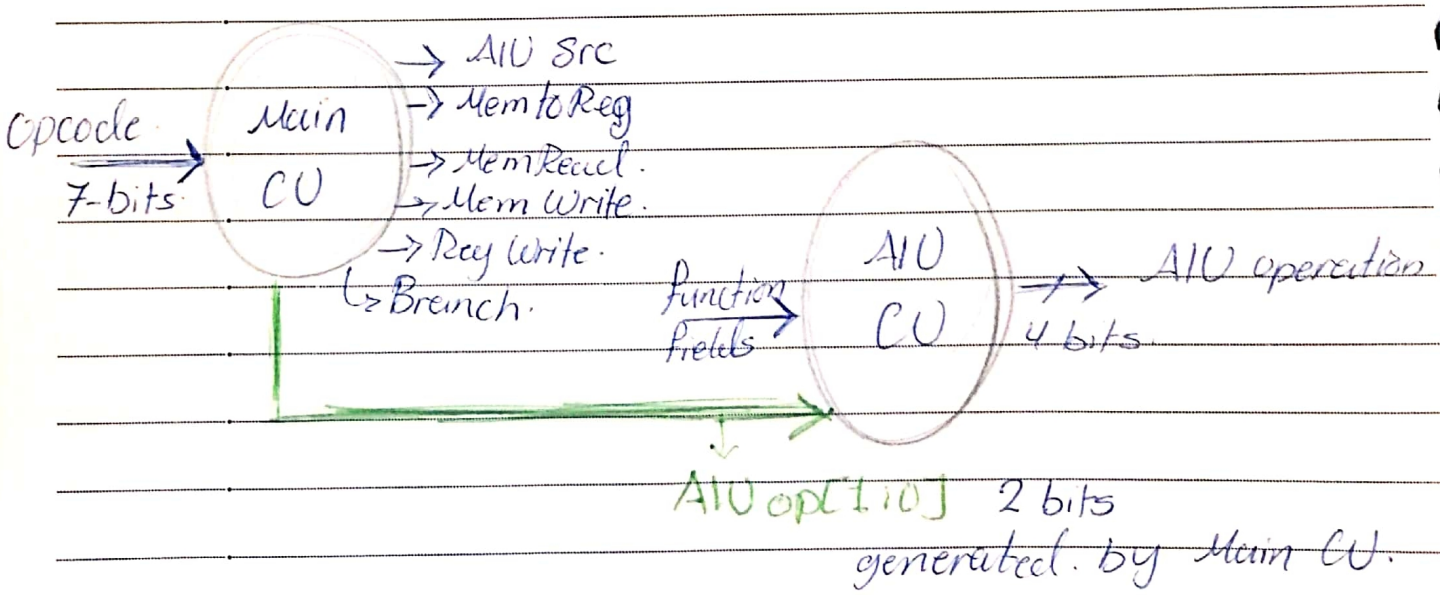
ALU Control:-



Note:-

ALU operations only depend on function fields for (ALU operation control bits [3:0])

Internal view:-



Scanned with CamScanner