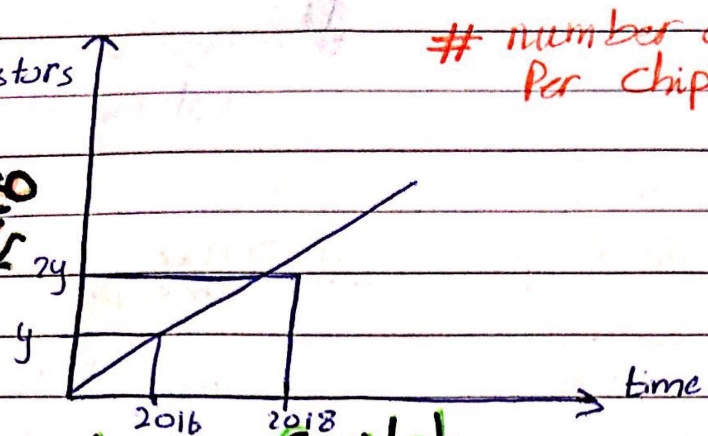
ORG

DR.WALEED DWEIK

BY:SARAH SA'ADEH

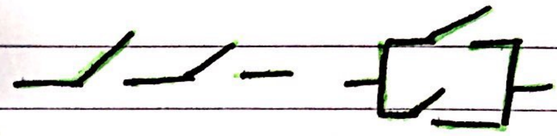
POWEROUNIT

* Moore's law *



number of transistors per chip (doubles) every 18-24 months

Transistor : Switch



* مع الوقت الـ Technology بتطور
بحيث انه الـ Trasistor size أو الـ Switch size
بصير أصغر تقريبا بقل للنص، لا يصغر للنص
بتضاعف عدد الـ Transistor بالمساحة

more transistor → more gates → more CPU (Processor)

* (Moore's law) ← على طول إدخال الـ computer in many applications ← smart Phones, cars, Search Engines

* Personal Computers (Laptop)
General Purpose → استخدامات متعددة
- run Simulation in MatLab
- web browsing
- Editing word file

* كل ما زادت الـ cost زادت الـ Performance.

* Server Computers:

High capacity , High Perf , High reliability

كل قديش بحزب او
كل قديش بعلق

- Network based

ال Server computer بجيه Request من ال user

Server → mail
→ web

↑↑↑↑

* زي موقع الجامعة بالـ إعلانات
مع أكثر من جدا وصول ال computers
مابصير بعلق .

* Super Computers:

تستخدم لل scientific applications اللي بدها power عالية

* مثال عليها ال Search Engine زي ال Google

، السرعة اللي بجيل فيها Search بتكون عالية وسريعة
عشان هيك بستخدموا ال super computers

* معظم ال super comp بتحتوي على 10^6 CPUs

* أعلى Capability مقارنة بال Personal/ Server / super

* لو قارنت بالـ overall comp market فهو اله أقل Fraction
لأنه بين الشركات الكبيرة محدد انه يكون بشركتها super computer.

⇓

Expensive كثير

وأعلى المبيعات ال Embedded computers / Personal comp

وأكثر استخداما

مهم #

* Embedded Computers
↳ competur Embedded (مُضمن) inside the System
→ Tv / cars / mobile phone ⟹ بكون ال Comp مُضمن فيها

- Software Already integrated with the device (hardware)
→ مابكون موجودة اعلاه بس بعمل install بس update.

Power ↓ (minimum Power) — قريش بصرف كهربا ء — Performance ↑ — Cost ↓

* الشركات الي بتنتج مبيعاتها smart phones اكثر من الشركات الي منتجاتها تشتغل على ال Pc

Pc Era — Pc
قبل لو كان عندي Pc comp بقدر اعمل عليه كل شي
بس بنزل ال Software على CD وبعمل Run بدون لا استخدم ال internet اذا ماكنت بدي اعمل Search

Post PC Era — PMD (mobile device) Smart Phone tablet — Cloud comp
بكون عليهم Software ولازم يكون عندي internet connection عشان اعمل Run

* Performance :-
- Algorithim

Task → 10 sec / 5 sec
ايهما أثر على ال Perf
كل ماقل الزمن بزيد ال Perf

- Language, Compiler, ISA

طريقة كتابة ال code

High Level language
C, C++, Java
↓
Assembly
↓
machine
0, 1

عملية التحويل ال language عند تنقل

Code
100 instructon — 200 instruction

المسؤول عن تحويل اللغة

ISA:-
add, sub — add, sub, divid, mult

ما رح يتم استعماله اقل

↓
عمل instructions

6/2/2020

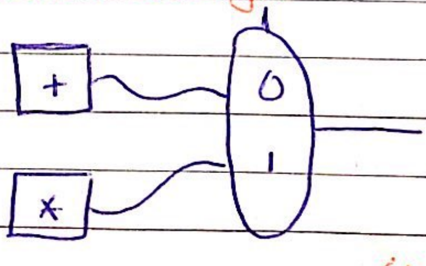
- Design for moore's law

Start 2y —— 2 years —— End 2y

(الهدف من عمل ال idea التبشير لتسريع عملية ال Design)

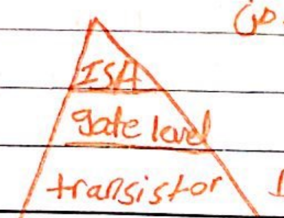
- abstraction
↳ Try to hide low level Design

لو حد طالب نعمل Design Add / mult



→ هون انا ما فتشت بال Design Details ال add وال mult

* لانه لو كل مرة فتشت بال Details هاي رح يصير ال Design تاخد كثير معقد وياخذ وقت كثير



عشان ازيد من سرعة ال Design

- make common case fast

بدي أخذ الأشي او العملية الي بتتكرر كثير وبدل اخليها سريعة
وهيك رح يتحسن الاداء

CPU₀ → Addition: 2 Sec, 95% ; divison: 20 Sec, 5%

⟹ CPU₁: add 1sec, divide 20sec ; CPU₂: add 2 sec, divide 15 sec

100 operation : $95 \times 2 + 20 \times 5 = 290$ sec

CPU₁ ⟶ $1 \times 95 + 20 \times 5 = 195$ Sec

CPU₂ ⟶ $2 \times 95 + 15 \times 5 = 265$ sec

← هاد احسن لانه قلل الوقت

CPU₀ → addition: ~~2sec~~, 80% ; divison: ~~20~~, 20%

⟹ CPU₁: add 1sec, divide 20sec ; CPU₂: add 2sec, divide 15sec

100 operation : $80 \times 2 + 20 \times 20 = 560$ sec

CPU₁ ⟶ $2 \times 80 + 20 \times 20 = 480$ sec

CPU₂ ⟶ $2 \times 80 + 15 \times 20 = 460$

* عشان اقدر اقرر لازم احسب ال Time بس
In general لو حسنت ال common case
رح يعطي perf احسن

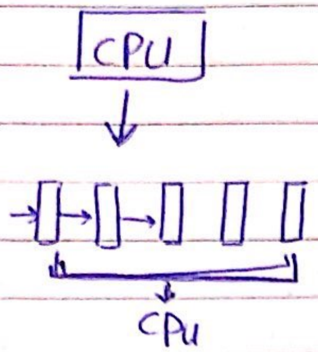
- Parallesim

Computer A: CPU

Computer B: CPU₀, CPU₁, CPU₂, CPU₃

* بقدر اعمل أكثر من instruction بنفس الوقت
او حتى بقدر اجزء ال instruction

- Pipelining (special type of Parallelism)



لما افلص ال Slice بتجدد
ابلش ب Operation جديدة

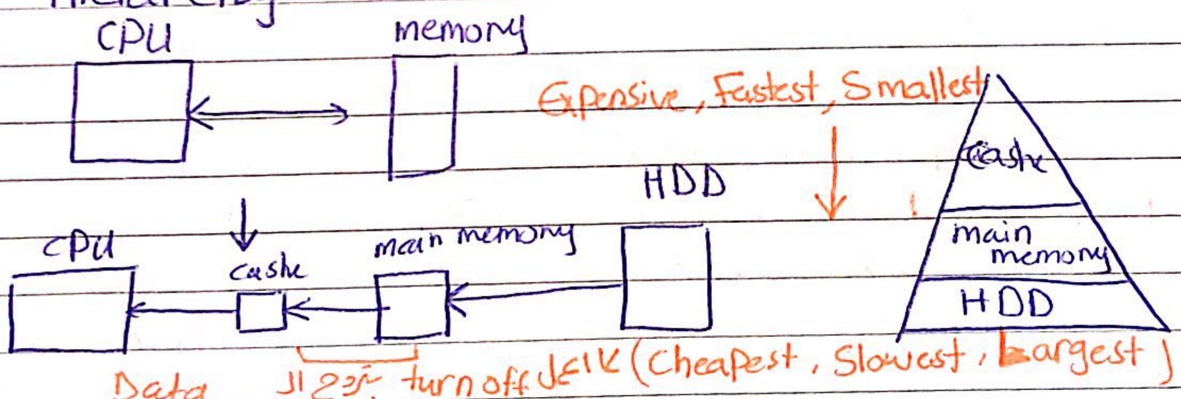
- Prediction

if ( )

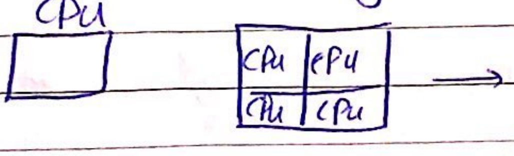
else

عشان احدد اي ... رح
استخدم رح يكون عندي وقت عالفاضي
عشان هيك الكمبيوتر تنبأ اي واحد رح يطلع صح
عشان هيك كان افضل نسبة
التنبؤ ... عالية

- Hierarchy



- Dependability → ما ... كتير



نجدد افضلهم
Perf وال Spare
ما رح تتغير

Three layers

High level language — System Software — Hardware

The advantage of using high level language :-
- using your own language   C=a+bi
- independent from hardware

C=a+bi

IBM — Intel

System Software

compiler — OS

Translate HLL to mashine code

بعمل تبادل task

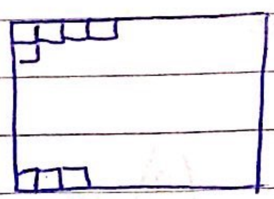
بنظم ال application system

هي بين ال layers اسمها interface

interface (System Software - Hardware) (ISA)
interface (Application Software - system Software) (ABI)

9/12/2020

LCD   مستخدمة لعرض ال images



كل pixel بتكون الها كود معين بال binary

* مجموعة ال pixels الي بيكونوا ال image اسمهم (Bit map)

ال size تبعت ال bitmap بتعتمد ع
1) Screen Size
2) ~~resolution~~ resolution

Buffer ≡ memory → مسؤول عن تخزين قيم ال pixels
↳ بتكون طابقة ال image

↓ raster refresh buffer (Frame buffer)

لازم ال bitmap size تكون تساوي ال size (buffer)

* Different value gives different color.

عملية ال refresh بتخليني اشوف الالوان لما تتغير بسرعة كبيرة بحيث اني ما الاحظها

* (Post PC Era ⟹ Touch screen)

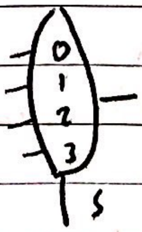
types of touch screen

- Resistive
- capacitive
  هاد اللي مستخدم اكثر
  لانه بتقدر اعمل multiple touches (zoom)

- Inside the Processor (cpu)

- Data Path

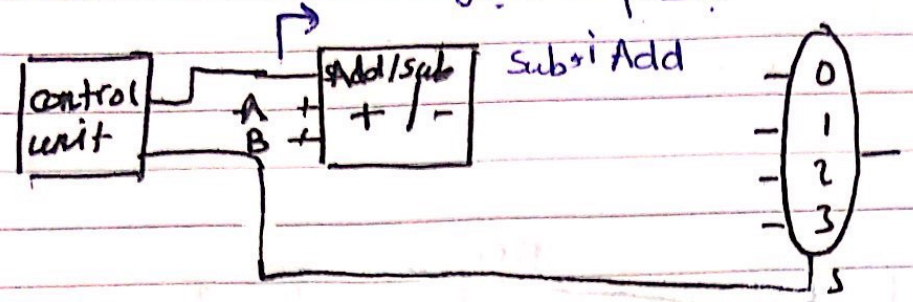
Processing لانه بعمل العمليات | +/- | data Path element



Select line

- Control

بتتحكم عشان يعمل Add أو Sub



* بناءً على شو ال control unit بتحدد شو بده يختار Add او Sub ؟
- حسب نوع ال Operation اي بده يعمل ال cpu

مثلاً لو بده يعمل Add ال Control unit بطلع قيمة (zero) عشان يحكي لل Adder/Sub انه يعمل Add

هاد كله موجود على onechip

A5 ↑

Cashe memory

* ال Cashe موجودة بال A5 عشان السرعة حيث بتكون better perf وبتكون كثير صغيرة

* Volatile main (primary) memory

* ال Data بتروح لما أطفي الكمبيوتر turn off

الطريقة الي بتتخزن فيها ال Data في ال Main memory هي using capacitors ← وهاد سبب انه ال Data بتروح

* فائدة استخدام هاد النوع اني بقدر اخزن Data كثيرة

هاد مستخدم بين ال Main memory وبال cashe ← SRAM

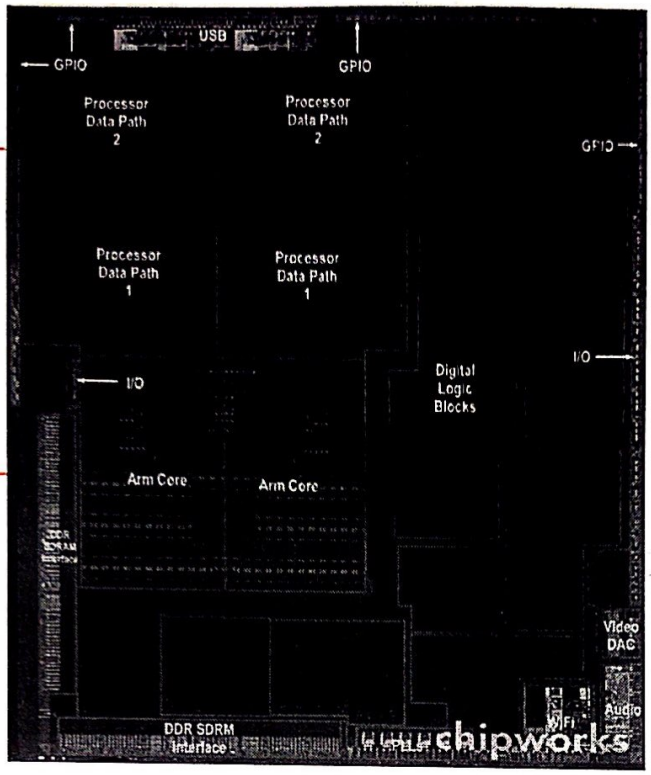
1 bit وعشان اخزن

بتحتاج size أكبر

2inverters , 2 transistors

Inside the Processor

Apple A5



الرسمة اسمها (Layout)

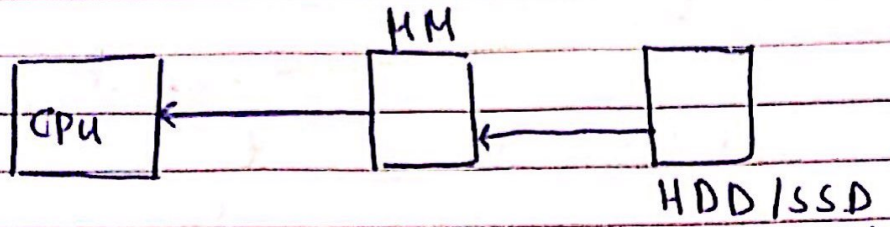
بتوضح توزيع الـ Parts بالـ integrated circuits

Graphical Prossesing unit (GPU) للأشياء اللي بتتعرض على الشاشة

Dual core

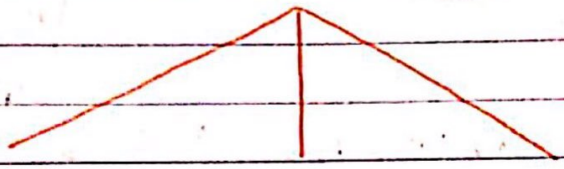
Tablets معظم الـ* و الـ Smart Phone بتعتمد على الـ Arm * والرقم لحقة خاصة

* Non - Volatile Secondary Memory
- Magnetic Disk (HDD)
- Flash memory
- CDROM - DVD



OS
استخدام هاد اسرع من ال HDD

Network

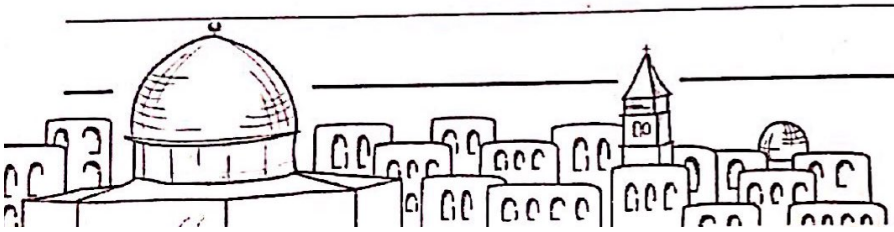


Local Area Networks
- Ethernet

Wide area Network
- optical Fibre

wireless
speed ال
تاستخدم نقل اشي

reliability ال والسرعة اكبر
وهاد اللي تنعم أكثر



11 - 2 - 2020

Responsetime

* كم الوقت الذي بيستغرق عشان أعمل task

* كل ما كان الزمن اقل كل ما كان الاداء احسن

Throughput

* كم task بقدر اعمل خلال وقت محدد 10^6 transactions/sec

* كل ما زاد ال throug put كل ما كان احسن

1) [CPU] ⟶ [CPU] faster
i5 → i7

- responsetime → decreasing
- Throughput → increasing

2) [CPU] ⟶ [CPU] [CPU]
i5 → i5, i5

- response time → Stays the same
- Throughput → increases

Performance = 1 / responsetime = 1 / ET
(ET: Execution time)

x is n-times faster than y

$$Perf\,x / Perf\,y = n \longrightarrow \frac{1/ET_x}{1/ET_y} = n$$

$$ET_y / ET_x = n$$

idle time

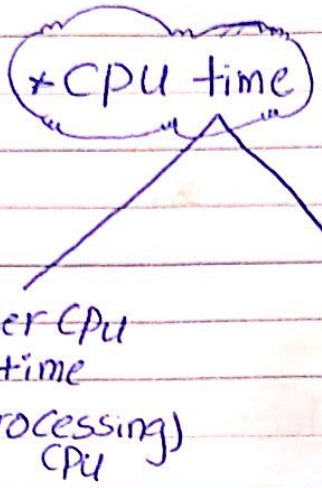
CPU → Program x
CPU → Program y

لما يشتغل ال Program x

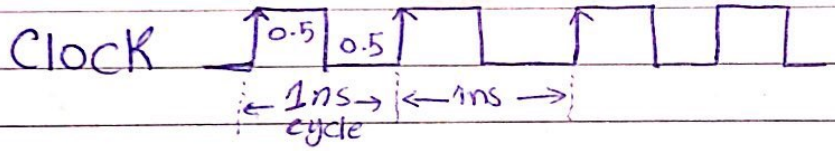
Program y بضل يستنى

ال Program x وهاد ال idle time

* Response time ≡ ET ≡ Elapsed time ≡ wall clock
include (Processing (CPU), I/O, OS overhead, Idle time)



- CPU Perf = 1/user cpu time



Clock Signal

$\frac{1\,sec}{1\,ns} = \frac{1}{1\times10^{-9}} \times 10^{9}$ cycle/sec

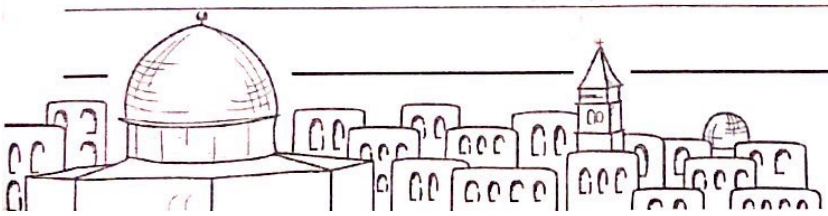
(F) Clock freq (number of cycle in 1 Sec)

(T) Clock period (Clock cycle time) = 1 ns

$F = \frac{1}{T}$

$F = \frac{1}{T} = 2\,GHz$

$T = \frac{1}{F} = \frac{1}{2\times10^{9}} = \frac{1}{2} \times 10^{-9}$ (0.5 ns)



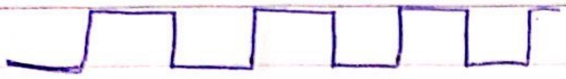
- CPU time = CPU clock cycles X Clock Period (T)

$$= \frac{\text{CPU Clock cycles}}{\text{Clock Rate (F)}}$$

To improve the Perf

- اقل عدد ال cycles او ال ClockPeriod
- او ازيد ال Clock Rate

3X2 = 6 ns



←2ns→

لا اطول اشي ال Clock cycle

تلقائي الزمن رح يتغير بس لازم اوصل كل وقت مناسب

2.5 ns

2.5 X 2 = 5 ns ↑ هاد افضل

13 - 2 - 2020

* CPU Clock cycles = Instruction count X Cycles Per instruction

لـ عدد ال instruction (IC) — لـ عدد ال cycles

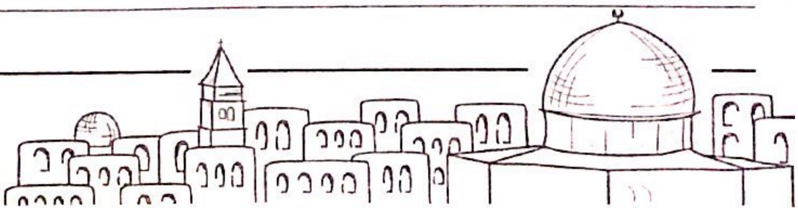
* CPU time = IC X CPI X T = $\frac{IC \times CPI}{Rate\ (F)}$

IC Depend on → ISA, compiler, Program

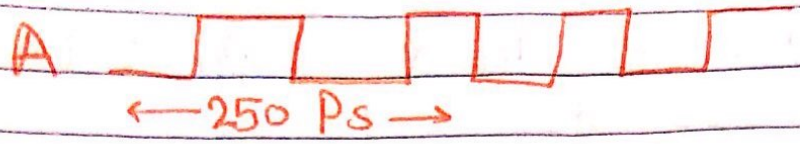
CPI Depend on → CPU hardware (يعتمد ع طريقة ال Design)

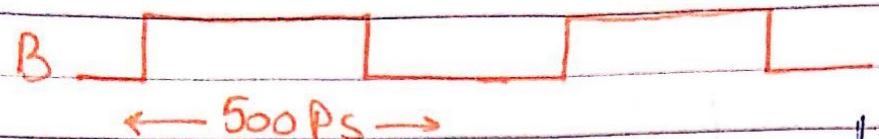
T Depend on → كل ما استخدم tech تبصير العملية اسرع، وبصير ال Design احسن

↳ يعتمد ع ال Technology



Scanned with CamScanner





- CPU time = $IC_A \times 2 \times 250 \times 10^{-12}$ sec
- CPU time = $IC_B \times 1.2 \times 500 \times 10^{-12}$ sec

CPU time A = $IC \times 500 \times 10^{-12}$

CPU time B = $IC_B \times 600 \times 10^{-12}$

$$\frac{\text{CPU time B}}{\text{CPU time A}} = 1.2$$

- A is 1.2 time faster than B

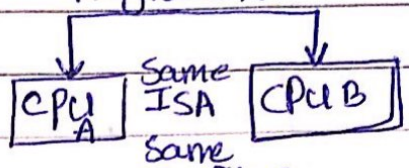
Note

Same ISA, compiler

↓

Same IC

Program x



Same ISA, compiler لأنها نفس الـ

Same IC لأنها نفس الـ

* $CPI_{Avg} = \frac{\text{CPU Clock cycles}}{IC} = \sum_{i=1}^{n} IC_i \times CPI_i / IC$

n ← number of instructions types (add, sub)

Clock ← عدد الـ Clock

Per Instruction

الـ CPI لنفس النوع

$= \sum_{i=1}^{n} CPI_i \times \frac{IC_i}{IC_{total}}$

Relative freq of instruction "i"

Example:

- Program x : 2 Add (i=1), 6 Sub (i=2), 10 mult (i=3)

$IC_1 = 2$, $IC_2 = 6$, $IC_3 = 10$ $IC_t = 18$

add → 2 cycles
sub → 3 cycles
mult → 6 cycles

- Program y : 4 add, 5 Sub, 2 mult

$IC_1 = 4$, $IC_2 = 5$, $IC_3 = 2$ $IC_t = 11$

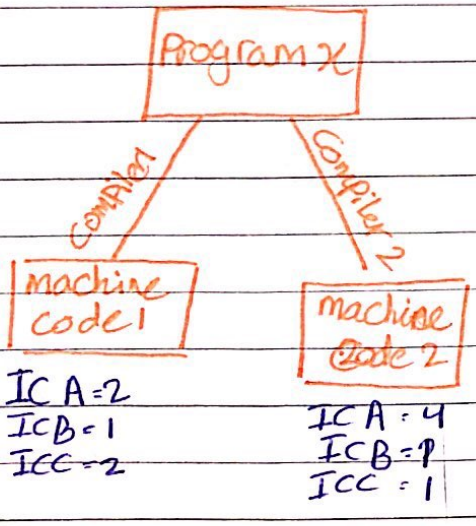
$CPI_{Avg} = \frac{2+3+6}{3} = \frac{11}{3}$

Sol:

$$CPI_{avg\ x} = \sum_{i=1}^{3} \frac{IC_i}{IC_t} \times CPI_i$$

$$= \frac{IC_1}{IC_t} \times CPI_1 + \frac{IC_2}{IC_t} \times CPI_2 + \frac{IC_3}{IC} \times CPI_3$$

$$= \frac{2}{18} \times 2 + \frac{6}{18} \times 3 + \frac{10}{18} \times 6 = \frac{4}{18} + \frac{18}{18} + \frac{60}{18} = \frac{82}{18} = 4$$



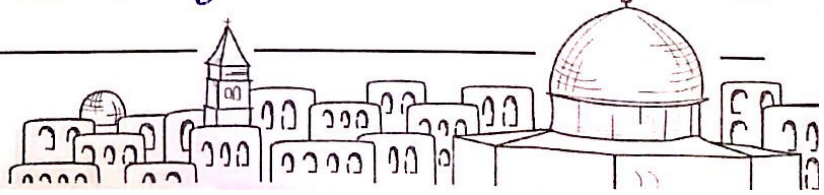
$CPU\ time = IC \times CPI_{Avg} \times T$
$= 5 \times 2 \times T = 10T_1$

$CPU\ time = 6 \times 15 \times T_2$
$= 9T_2$

لو نفس الـ CPU و يكون الثاني أسرع

- $CPI_{Avg} = \frac{2}{5} \times 1 + \frac{1}{5} \times 2 + \frac{2}{5} \times 3 = 10/5 = 2$

- $CPI_{Avg} = 4/6 \times 1 + \frac{1}{6} \times 2 + \frac{1}{6} \times 3 = 9/6 = 1.5$



Class	A	B	C
CPI	1	2	3
relative freq Sq1	40%.	20%.	40%.
relative freq Sq2	68%.	16%.	16%.

$$CPI\ Avg = \frac{40}{100} \times 1 + \frac{20}{100} \times 2 + \frac{40}{100} \times 3$$

18 - 2 - 2020

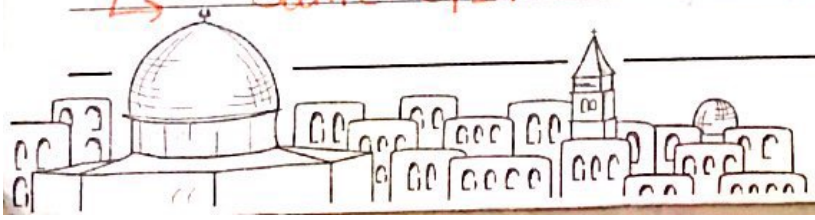
$$cpu\ time = cpu\ clock\ cycles \times Time$$
$$= IC \times CPI_{Avg} \times T$$

$$\left(\sum_{i=1}^{n} IC_i \times CPI_i\right) \times T = IC \times \sum_{i=1}^{n} \frac{IC_i}{IC_t} \times CPI_i \times T$$

↳ Relative freq

#

* To compare two cpus, we only cpu time or cpu Perf
* ISA + Compiler are the (Same IC)
* Same Comiler (Hardware)
  - Same "T" or "f"
  - Same CPI:



Program X

Compiler 1 | Compiler 2

Seq 1	Seq 2
ICA = 1	ICA = 2
ICB = 2	ICB = ?
ICC = 4	ICC = 2

→ Hardware

$$\frac{\text{CPU time}_2}{\text{CPU time}_1} = 2 \leftarrow \frac{\text{Slower}}{\text{faster}}\ (\text{time}) \quad : \quad \frac{\text{faster}}{\text{slower}}\ (\text{Perf})$$

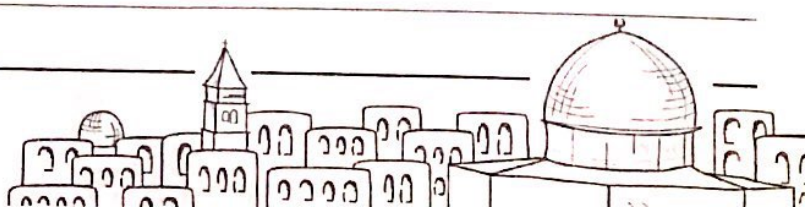
$$\Downarrow$$

$$\frac{IC_2 \times CPI\,Arg_2 \times T_2}{IC_1 \times CPI\,Arg_1 \times T_1} \Rightarrow = \frac{\sum_{i=1}^{3} IC_i \times CPI_i}{\sum_{i=1}^{3} IC_i \times CPI_i}$$

$$= \frac{2\times1 + 7\times2 + 2\times3}{1\times1 + 2\times2 + 4\times3} = \frac{8+2\ ?}{17} = 2$$

much faster — صير القانون اي شي

the best Perf measure → Execution time



Chapter 2 :

* الـ Hardware متشابه لمعظم الـ [redacted] الـ Computers لأنهم كلهم بستخدموا نفس الـ Technology.

RISC	CISC
- Reduced instruction set computers	- complex instruction set computers
- the same length	- variable length
- fast execution	- slow execution
- simple functionality	- complex functionality
عدد الـ instructions أكبر	Example : Intel X86

* معظم الـ smart Phones بعتمدوا على الـ RiSC ، op بتحتاج ذاكرة أكثر

* في الـ server لسا بستخدموا CiSC

Design Principle 1

hardware simple

* كل ما يكون الاشي Regular ع تخلي الـ implementation سهلة

⇓

high Perf

Lower Cost

rd rs1 rs2

add a, b, c ⟹ $a = b + c$

destination Sources

مش كثير مهم الترتيب بال add

sub a, b, c ⟹ $a = c - b$

destination source

* مهم الترتيب

$a = b + c + d + e$; ⟹ high level language

compiler2 compiler1

2 different compiler

compiler 2	compiler 1
add a, b, c	add t_1, b, c;
add a, a, d	add t_2, d, e
add a, a, e	add a, t_1, t_2

* بنقدر نخلي ال Source يكون destination

another Solution Example slide 6:

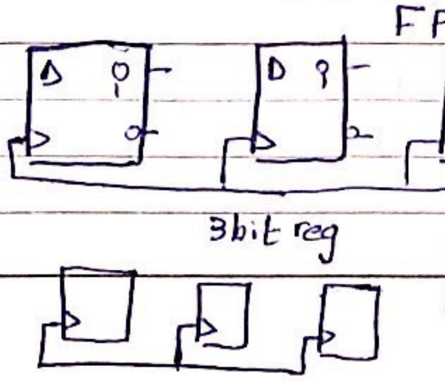
add f, g, h
sub f, f, i
sub f, f, j

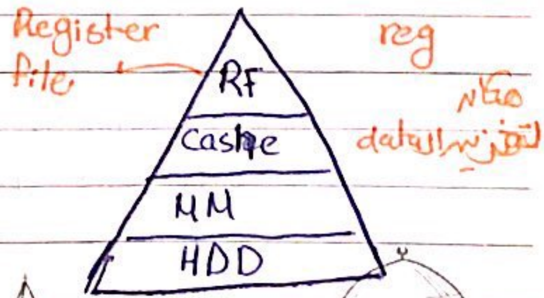
* ال Add وال sub غالبًا الهم نفس ال complixity

Design principle 2:

ال memory كل ما تكبر كل ما تكون ابطأ

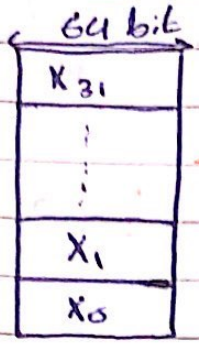
- Smaller is faster





Risc - V 64 bit

Risc - v 32 bit



→ use for freq accessed data

byte 8-bit

word 32-bit (4bytes)

double word 64-bit (8bytes)

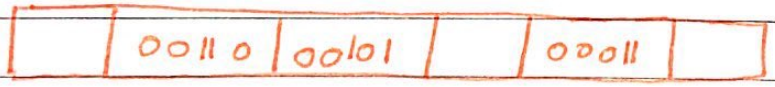
* ليش Size الـ Reg 32 bit ؟

كل ما كان الـ Design كل ما اسرع

high Perf

✓ execution time

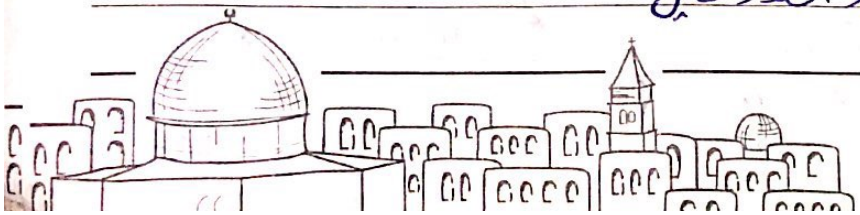
Add X_3, X_5, X_6



* كود كتاج 15 bits عشان اقدر شوف الـ instruction وافضل عشان high Perf ← operation الـ عشان 17 bits

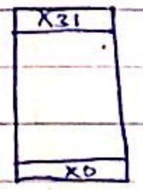
Smaller is faster

* لو اقرت 9 bits عشان اقلل كل الـ operation عشان اقدر شوف الـ instruction و احتاج 27 bits وافضل عندي 5 bits وهاد العدد قليل



20/2/2020

* Design Principles
- Simplicity favors regularity
- Smaller is faster



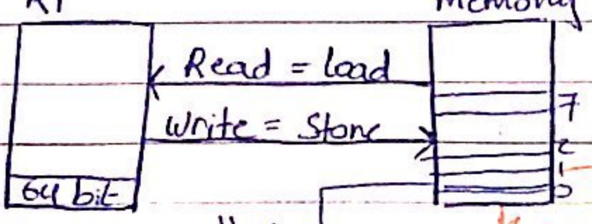
* Arithmetic operations use register operands

add X_{19} , X_{20} , X_{21}
Sub X_{19} , X_{19} , X_{22} ← مناصبة Perf نفس الشي *
Sub X_{19} , X_{19} , X_{23} ← هادا ستخدم Reg اقل فهو احسن *

int A [10]

⋮

C = g + A[5] → مخزنة بالـ memory فلازم الاقي طريقة
علشان استدعيها ، الـ RISC-V بتعطيني
instruction علشان تنقل الـ data من الـ memory
الـ Register

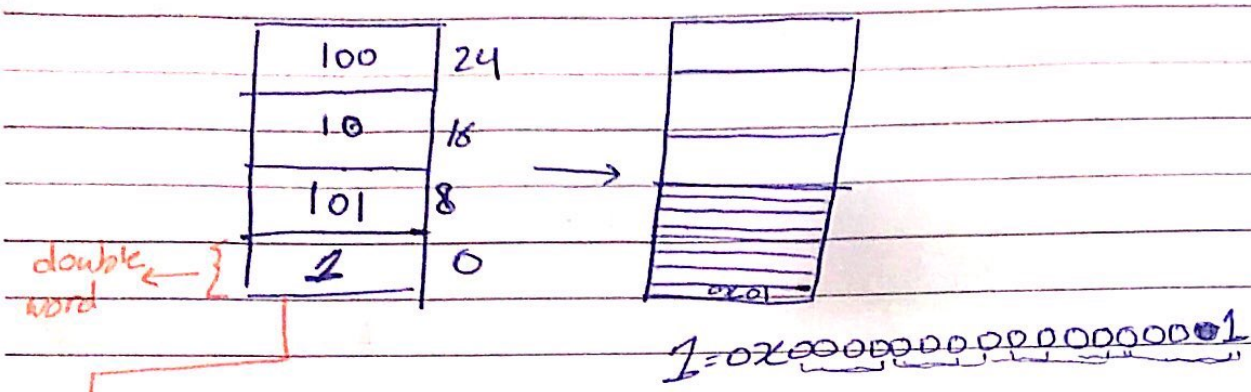


للـ memory عنى مقسمة
8 locations بحز
8 locations

طريقة تخزين
تكوين انو الـ memory مقسمة
chips وكل chip حجمها 8-bit

* الاسماء اتاروها based على
الـ Memory
64 bit / 8 bit = 8 locations

* لو بدي اقرأ من الـ memory بحفظ أول Address
أول byte بجيب هو بقرأ الباقي



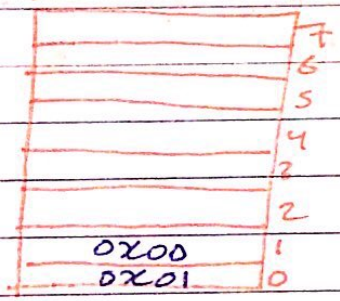
Little Endian → RISC-V

↓

least Address

least Significant بخزن ال

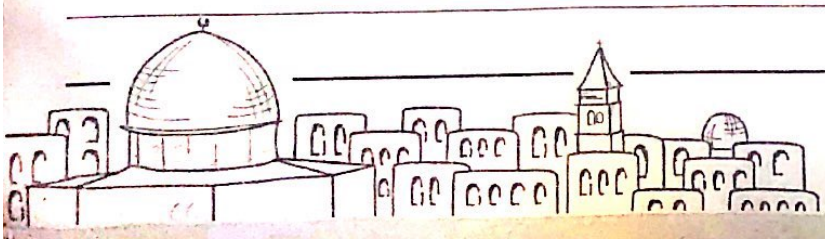
byte



$(1)_{10}$ = 0000 ... 00 0001

0x0000000000000001

* يعني ← (0x) الـ
Hexadecimal



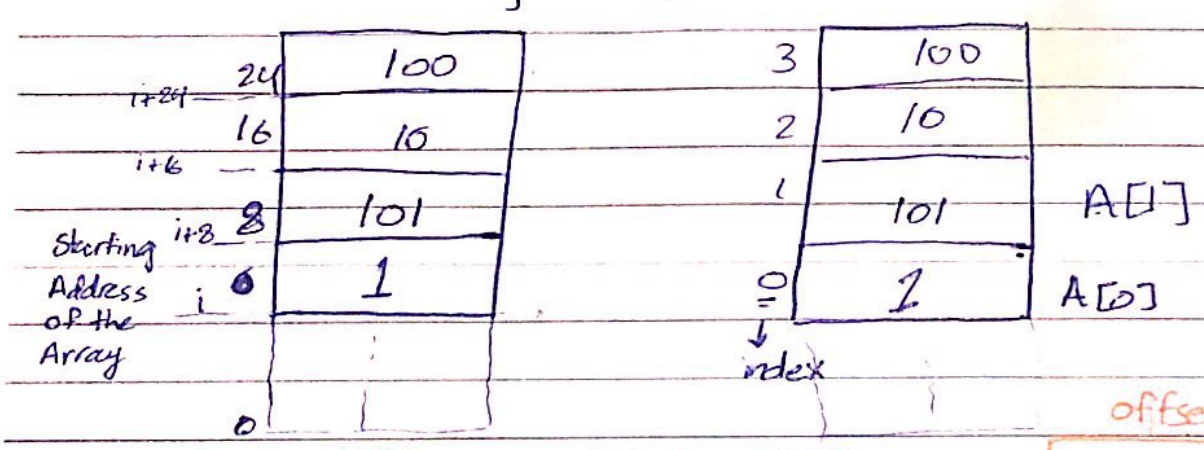
int A[10];

C = g + A[5]

↓

index (0 - 9)

*كيف بدي اعرف ال A[5] وين موجودة بال memory؟



byte Address = Starting Address of the array in the memory + (index * data size in byte) ⇓ offset

8 byte

long long int B[5];

long long int A[9] ⇓

= i + 1 X 8 = i + 8

*Load double word ⟹ Ld X_{28}, 32 (X_{18})

destination ← ↳ source

← Memory [X(18) + 32]

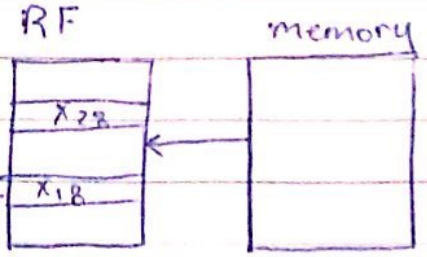
(X_{28})

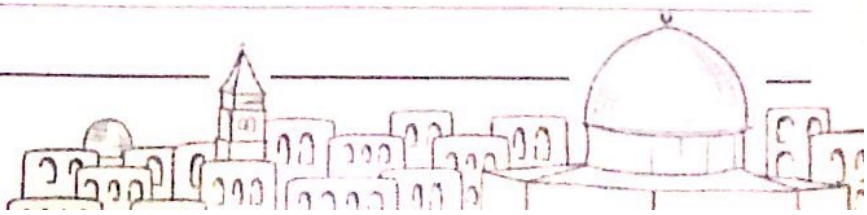
Address

Starting Address

↓

Read



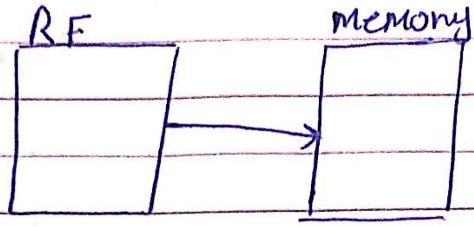


Store double word ⟹ Sd X_{28} , 32(X_{18})

(بقرأ منه مابكتب عليه) source ← (under X_{28}) ; source (under X_{18})

مش destination

Source
يعني مابعدل
عليه



23 - 2 - 2020

* Ld X_{28} , 32 (X_{18})
  - X_{28} ↓ destination Reg
  - 32: offset
  - (X_{18}) ↳ base reg = contains the starting address at the array in the memory

* Memory address = Starting address + index X data size in bytes

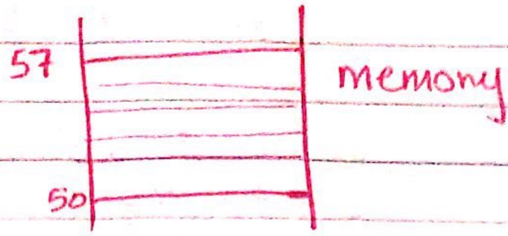
← Memory [32 + (X_{18})]

* Sd X_{28} , 32 (X_{18})
  - X_{28}: source
  - 32: offset
  - (X_{18}) ↳ base register

ld X_{25} , 50 (X_0)

Memory [50 + (X_0)]

(X_{25}) ← Memory[50]



* Absolute Address

- بيكون عبارة عن رقم محدد بيضع كل الـ offset قيمته ع الـ base reg X_0

\+ Relative address

- بتعامل بالـ Array ما بيكون عبارة رقم الـ offset بالزبط

g = h + A[8]

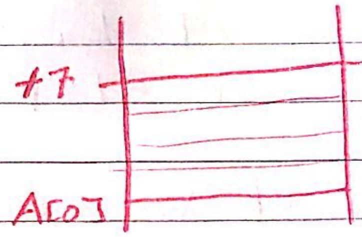
- اذا Assignment statement بيستخدم Array ع الـ Right side بيكون load

- اذا ع الـ left side بيكون store

* each element of A is 64-bit (8byte)

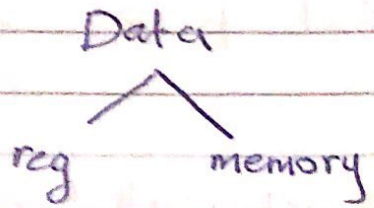
- ld X_5 , 64 (X_{22})
- add X_{20}, X_{21} , X_5

↳ offset = index X data size
= 8 X 8 = 64
↳ (8bytes)



• A[12] = h + A[8]

- Ld X_5, 64(X_{22})
- add X_5, X_{21}, X_5
- sd X_5, 96(X_{22})
  ↳ 12x8 = 96



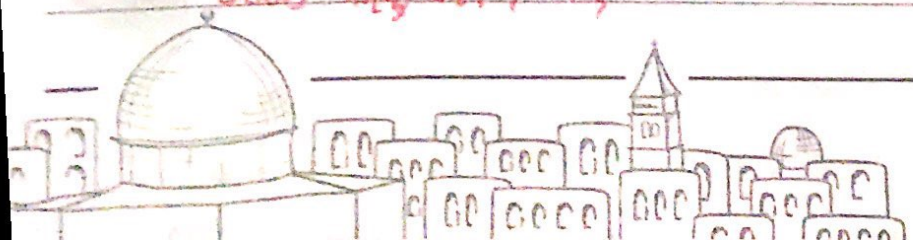
- much faster
  اسرع من الـ memory
  واقرب الـ cpu
- less energy

الاشياء اللي دايما بعمل عليها
Reg تكون بال ld, sd
وهاد الاشي مسؤولية الـ compiler

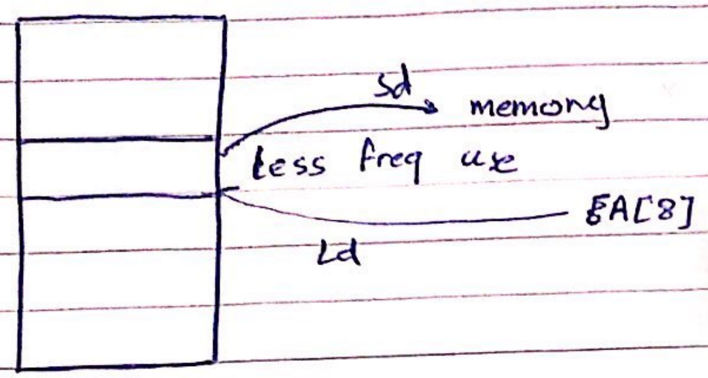
A[12] = h + A[8]
g = h - A[8]

Ld X_5, 64(X_{22})
add X_6, X_{21}, X_5
sd X_6, 96(X_{22})

Sub X_{19}, X_{21}, X_5



Register spilling

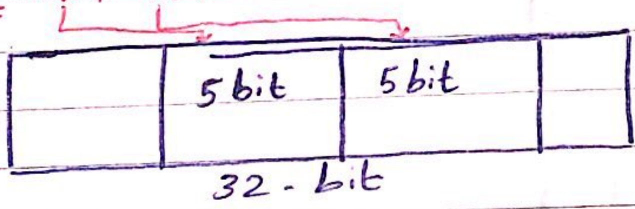


for loops (i++ i--)   immediate = constant

A = C + 4 ⟶ بتكون مخزنة بعنوان Address وبدل ld بس من الحالة الخاصة ⟹ مثال على ال make common case fast

↓ X_{10} ↓ X_{11}

addi X_{10}, X_{11}, 4



addi X_{10}, X_{11}, -4 لانه بقدر أعمل sub immidiate ما عنا

- a = -b ; ⟹ negation
  ↳ X_{20} ↳ X_{22}

استخدامات ال X_0

↳ sub X_{20}, X_0, X_{22}

- Ld, sd absoulte addresses

X_{20} ↑ X_4 ↑

$a = 4 + c_i$

addi $X_{20}, X_{21}, 4$

$a = c - 4i$

addi $X_{20}, X_{21}, -4$

$a = 4 - c$

addi $X_5, X_0, \ldots 4$

Sub X_{20}, X_5, X_{21}

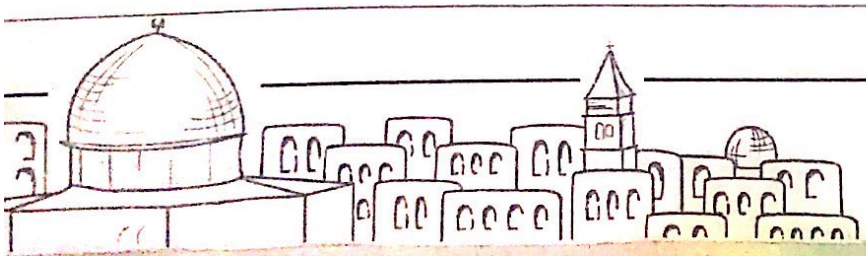
$$\underset{3}{\_}\ \underset{2}{\overset{i}{\underset{d}{\_}}}\ \underset{1}{\_}\ \underset{0}{\_}$$

value = $d \times r^i$

8	4	2	1
1	0	0	1
3	2	1	0

n bits $X_{n-1} \ldots X_2\ X_1\ X_0$

Values $X_{n-1} \times 2^{n-1} + \ldots X_0 \times 2^0$



0 to $2^n - 1$ → range

0000 ← ↵ ↳ 11111

الصفر أكبر

64 bit unsigned

0 to $2^{64} - 1$

111 --- 111 ← ↱ 11 --- 1111

add X_{28} , X_{28} , X_{24}

هون رح يصير over flow وهاد الإشي الـ OS

بتعرف منه

25-2-2020

addi , , -4

addi , , +4

0 → Positive 1 → Negative

Representation for sign numbers

1) Sign - Magnaitude

2) 1's Complement

3) 2's complement

0 100 → +4

1 110 → -6

$$
\begin{array}{rl}
(+16) & 0\ 0010000 \\
(-24)^{+} & 1\ 0011000 \\
\hline
 & 1\ 0101000 \rightarrow -40 \quad X
\end{array}
$$

→ Sign mag → disadvantages
① ② two representation for zero

0001 (+1)
+ 1001 (-1) → sign magnitude
1110 (-1) → 1's complement

$1111 \Rightarrow 2^n - 1$

أكبر عدد +ve	0111	$\rightarrow (2^{n-1}-1)$
أقل عدد -ve	1000	$\rightarrow -(2^{n-1}-1)$

$X + (-X) = 2^n - 1$

* 2 representation for zero

2's Complement

$0\,1111 \cdots 11 = 2^{n-1} - 1$
$10000 \cdots 0 = -(2^{n-1})$

$+2^{n-1}$ → can't be represented

Single representation for zero

$(1111)_2 = (-1)_{10}$

$1\times2^0 + 1\times2^1 + 1\times2^2 + -1\times2^3$

$1 + 2 + 4 - 8 = -1$

* رح نتعامل مع ال 2's complement عند ال Hardware
لأنه بسهل ال Signed - Unsigned numbers

(+16) 0 0010000
(−24) 11101000
———————
1 1111 000
= −8

24
00011000
↓↓↓↓↓↓↓↓
11101000

$X + \bar{X} = 111 \ldots 11 = 2^n - 1 = -1$

$X + \bar{X} = -1 \Rightarrow$ [$-X = \bar{X} + 1$] (complement) → negation

Approach 1: ↗

$x + (-x) = 2^n$

$x + (\bar{x} + 1) = (x + \bar{x}) + 1$

$1111 \ldots 1 + 1 = 10000 \ldots \Rightarrow 2^n$

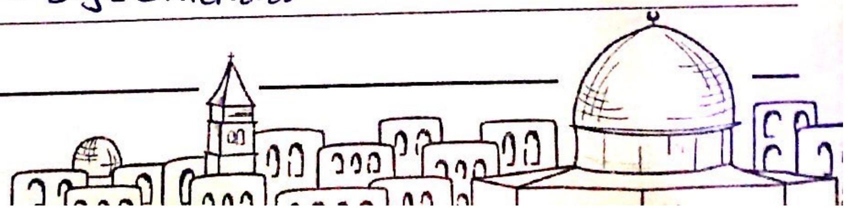
$-2 \rightarrow (1 \ldots 111110)_{64}$

$+2 \rightarrow 0 \ldots 000010$

add, Sub X, X, X → Sign extended لأنه بالأصل الرقم نفس ال size ال Reg

Load, Store X_2, offset(X) → sign extended (12 bit + 64 bit)

addi X_5, X_{10}, 5 → (64 bit + 12 bit) → sign extended



عدد الـ format قليل والهدف منه يكون regular

R - Format ⟹ Register - Format

الي تزبط هاد النوع اللي الـ operands تبعونه ← registers
(Add / sub) (And / or)

funct3	rs_2	rs1	funct3	rd	opcode
	5 bits	5 bits	3 bits	5 bits	7 bits

opcode → تحديد نوع الـ instruction

rd → destination

rd ← rs_2 / rs_1 جمع، طرح

مجال لـ opcode زيادة = funct3 / funct7

لانه لو بنشي بس 7bit هيكون عندي $2^7 = 128$

عمليات محدودة وكان لو بدنا عمليات جديدة

باستخدام الـ funct3 / funct7 ← 2^{17} =

27 - 2 - 2020

I - format ⟵ * Ld - addi
↳ immediate   sd → Format ثاني

↳ نفس الفورمات لانه الهم نفس عدد ال reg والوظيفة

لازم اعمل sign extended

- Ld rd, offset(rs1) ⟹ (rd) ← Memory[(rs1) + offset]
- addi rd, rs1, imm ⟹ (rd) ⟵ (rs1) + imm

immediate	rs1	funct3	rd	opcode
12bit (b_{31} … b_{20})	5bit (b_{19} … b_{15})	3bit (b_{14} … b_{12})	5bit (b_{11} … b_7)	b_6 … b_0

I - format

* اكبر رقم بقدر اعبه بال imm ⟵ $2^{11} - 1 = +2047$
011111111111

* أقل رقم ⟵ $-2^{11} = -2048$

* ال offset نفس ال imm بس الفرق انه موجع وطرح
هو إزاحة أكثر اشي 2047 bytes لفوق و 2048- bytes لتحت

Design Principle 3 :: Good design demand Good compromises

R - Format

imm	

- لكي يكون كل instruction الهم نفس ال format بين بالمقابل
اما أكبر الحجم من 32 ⟵ 64 ودح تقسير الذاكرة ابطأ
- او اخلي ال formats يشبهوا بعض الى حدا واخليهم نفس ال size

وهم اختاروا الخيار الثاني

S-format

Sd rs_2, offset(rs_1) Memory[offset + (rs_1)] ← (rs_2)

7-bit 5bits

imm [11:5]	rs_2	rs_1	$funct_3$	imm [4:0]	opcode
5bits			3bits	5bits	7bits

S-format
Sign extended

	rs_2	rs_1	Funct3	rd	opcode

R-Format

imm	rs_1	funct3	rd	opcode

I-format
Sign extended

A[30] = h + A[30] + 1

ld X_9, 240, (X_{10})

add X_9, X_{21}, X_9

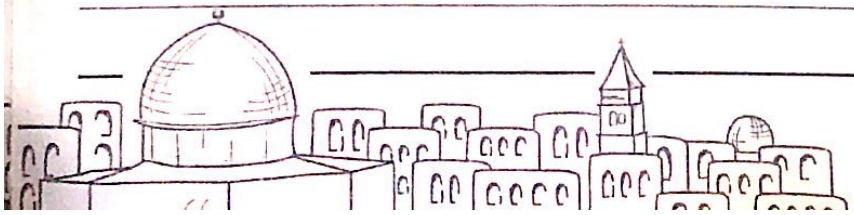
addi X_9, X_9, 1

Sd X_9, 240(X_{10})

I-Format

240	(10)	(3)	(9)	3
	01010	011	01001	0000011

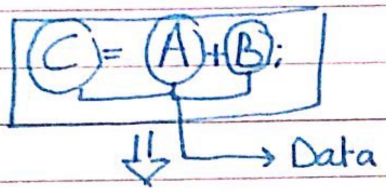
(0)	(9)	(21)	(0)	(9)	(51)
0000 000	01001	10101	000	01001	---



(1) = 000000000001	(9)	(0)	(9)	(19)	I-format
	01001	000	01001	----	

(7)	(9)	(10)	(3)	(16)	(35)	S-format

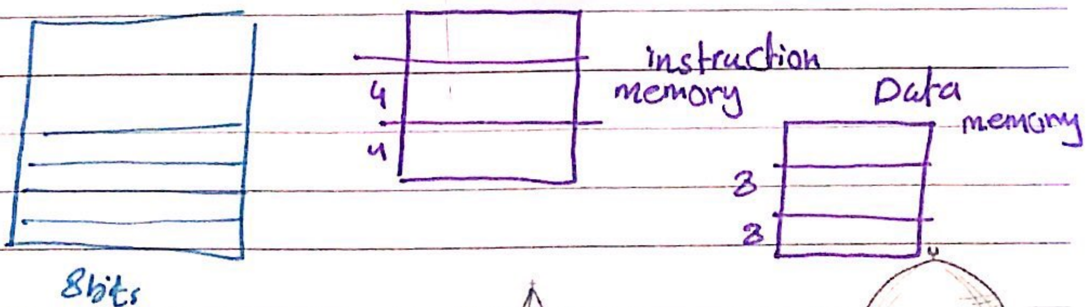
Instructions → (32 bit) → memory → جزء منها بكون الـ Data وجزء الـ instruction

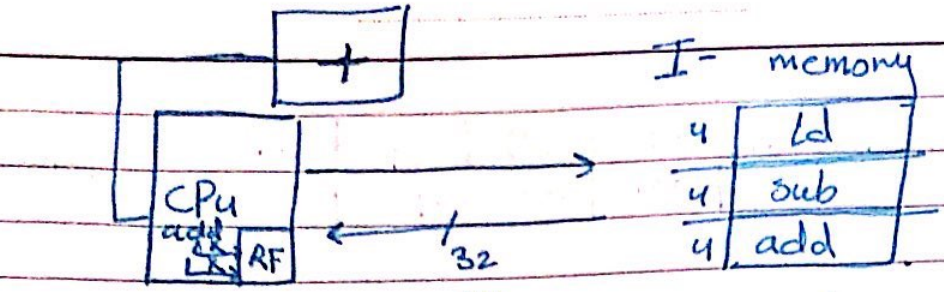


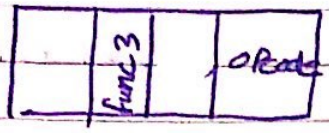
Data → memory, RF (64 bit)

* كيف الـ Instruction (32bit) والـ Data (64 bit) وهم الاثنين مخزنين بالـ memory ؟

الـ instruction بجزئها 4 bits والـ Data 8 bits







1 march 2020

Small numbers ← Slli X_5, X_3, 5 $\quad (X_3) \times 2^5$

X_3 | 0 ---- 000011 | $\Rightarrow (3)_{10}$

X_5 | 0 ---- 00011b0000 | $\Rightarrow (96)_{10}$

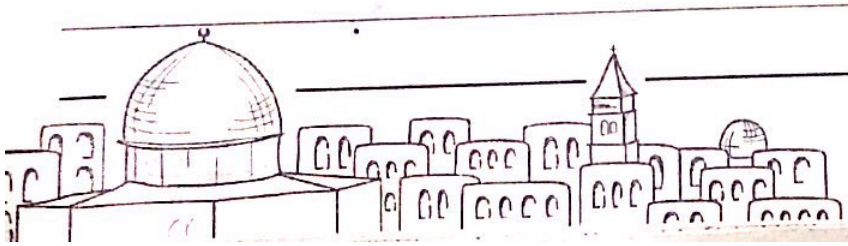
Srli X_6, X_7, 2

$(15)_{10} \Leftarrow X_7$ | 0 --- 000 1111

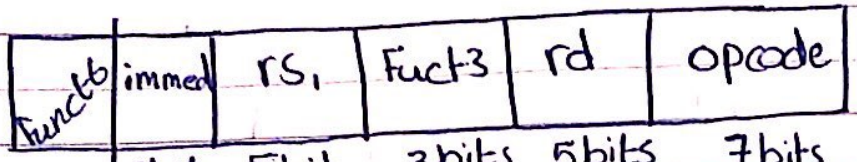
$(3)_{10} \Leftarrow X_6$ | 00 ---- ---- 0011

$(X_7) \div 2^2$

$15 \div 4 = 3.75$



Slti rd, rs1, imm
Srli rd, rs1, imm



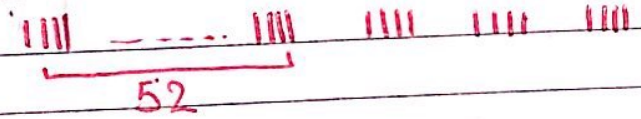
maximum shift (63bits)

0 ⋮ 63 } 2^6 64

And operations

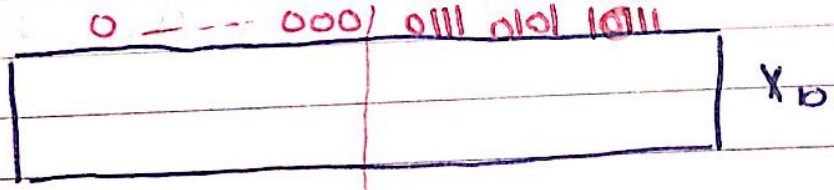
R-format and rd, rs1, rs2 ⟹ (rd) = (rs1) And (rs2)
I-format andi rd, rs, imm ⟹ (rd) = (rs1) And ↓ sign extended

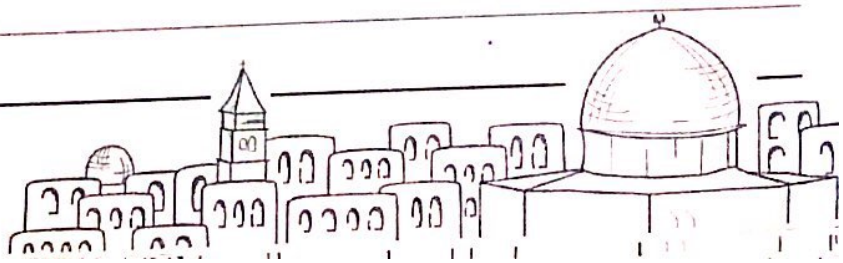
andi X9, X10, 0XFFF



andi X9, X10, 0X75B

0 --- 0001 0111 0101 1011





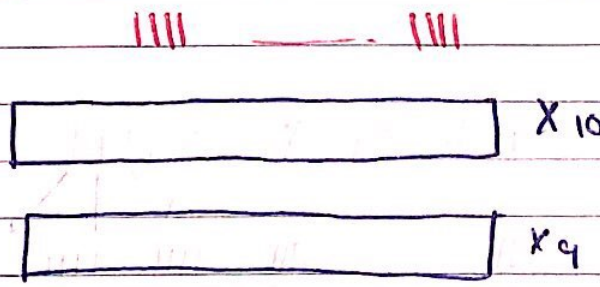
+ XOr operation

$C \oplus 0 = C$	
$1 \oplus 0 = 1$	
$0 \oplus 0 = 0$	
$C \oplus 1 = \bar{C}$	
$1 \oplus 0 = 0$	
$0 \oplus 1 = 1$	

Xor rd, rs1, rs2 R-Format
Xori rd, rs1, imm I-Format

Not ⟶ Xori

$X_9 = \overline{X_{10}}$ Xori X_9, X_{10}, 0XFFF

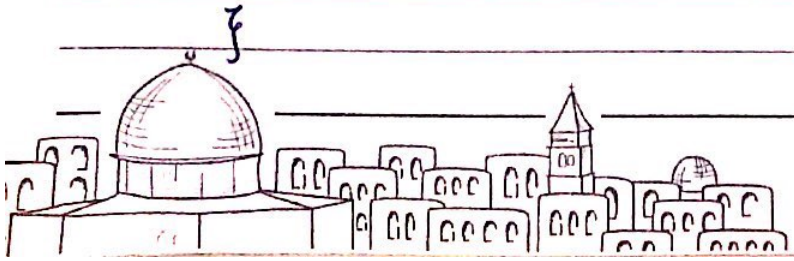


+ Conditional operations

if (condition)
{
}
else
{
}

Assembly

branch



$rs_1 = rs_2$ → condition true → take

beq rs_1, rs_2, L1 — false: $rs_1 \neq rs_2$ → NOT taken

branch ↵ ↳ equal — ↳ label

label: لازم يكون إلها instruction إلي بعدين اروحلها عشان يكون إلها

if (rs_1) == (rs_2)

bne rs_1, rs_2, L1 — (rs_1) ≠ (rs_2) → taken; (rs_1) = (rs_2) → NOT taken

branch ↵ ↳ not equal

X_{22} X_{23}

if (i == j)

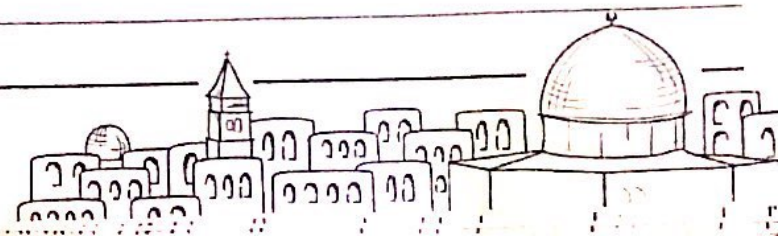
X_{19} f = g + h i → X_{21}

else ↳ X_{20}

f = g - h i

⇓

beq X_{22}, X_{23}, L1	bne X_{22} X_{23}, L1
Sub X_{19}, X_{20}, X_{21} i	add X_{19}, X_{20}, X_{21} i
beq X_0, X_0, Exit	beq X_0, X_0, Exit
L1: add X_{19}, X_{20}, X_{21} i	L1 sub X_{19}, X_{20}, X_{21}
Exit i ----	Exit. ----



3 march 2020

```
While (Save[i] = -K)        -K → X24
{
    i = i + 1;
}
```

```
Loop: Slli X10, X22, 3
      add  X10, X10, X25
      Ld   X9, 0(X10)   ⟹ (X9) = Save[i]
      bne  X24, X9, Exit
      addi X22, X22, 1
      beq  X0, X0, Loop
Exit: ——
```

offset = i * 8

(X10) = (X22) * 8 (23)

(X25) + (X22) * 8

branch if less than ≡ blt

blt rs1, rs2, label

rs1 < rs2

branch if ... greater than or equal

bge rs1, rs2, label

rs1 ≥ rs2

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

```
blt X23, X22, L1
beq X0, X0, Exit
L1: addi X22, X22, 1
Exit: ——
```

```
bge X23, X22, Exit
addi X22, X22, 1
Exit: ——
```

Compiler

Save [i] = = K

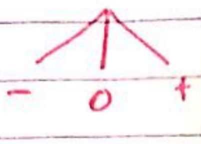
0 ≤ i ≤ size

Long Long int Save [10];

Save [12] = 15;

bgeu x_{20}, x_{11}, Index out of Bounds

↓ index ↓ size (+ve)



$x_{20} = 1$ —— $x_{20} = 0$ ——

$x_{11} = 0$ —— ✓ $x_{11} = 0$ ——

for (i=0 ; i<10 , i++)

Save [i] = Save [i] * 2;

x_{20} — base address register x_{25}

```
add x20 , x0 , x0   or   addi x20, x0, 0
addi x5 , x0 , 10
loop: bge x20 , x5 , Exit
slli x6 , x20 , 3          i x 8
add x6 , x6 , x25
ld x7 , 0(x6)              x7 = Save [i]
slli x7 , x7 , 1           x7 = Save [i] * 2
sd x7 , 0(x6)
addi x20 , x20 , 1
beq x0 , x0 , loop
Exit: ---
```

5 March 2020

```
Switch (n)          ⟶ long long int        if (n == 0)
{                                          {
case 0:                                      ---
        ===                                }
        break;                             else if (n == 1)
                                           {
case 1:                                      ---
        ===                                }
        break;                             else
                                           {
default: ---                                 ---
}                                          }
```

⇓

```
bne X20, X0, case1
   ---
beq X0, X0, Exit
case1: addi X5, X0, 1
   bne X20, X5, Case2
   ---
   beq X0, X0, Exit
case2:

case98: addi X5, X0, 98
   bne X20, X5, default
```

*لكل case بحط 3instruction زيادة.

Basic Block:

L1: - - - -
- - - -] → instructions Set
- - - -
- - - -
⋮
beq

They are executed All together or not

مش كل التنفذ
جزء منه
instruction ال
كان عنه ماشي
branch لـ label
الإجابة الأعلى

Procedure (function) calling

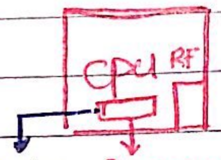
F = Average (5, 7, 10);
↳ Function call

$X_{10} \rightarrow X_{17}$ to place Parameters

* Jump and Link = jal

jal rd, Label

X_1 لإسترجاعها

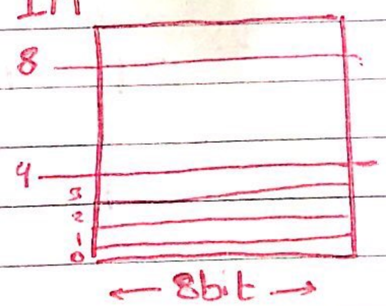


حجم البرنامج

$(0 - to - (2^{64} - 1))$ bytes

64 bit Program counter (PC) initialy 0 + 4

Instruction memory IM



نطلع منه = 4

Adress ال memory ال

الي عند 4

PC	
0	add
4	sub
8	— —
12	
16	jal rd, label → Jump to the target address link ← L→ 16+4
20	

beq X_0, X_0, Exit ≡ jal X_0, Exit] unconditional Jump/branch

jump and link register = jalr

jalr rd, offset(rs1)

link ← L→ (offset + rs1)

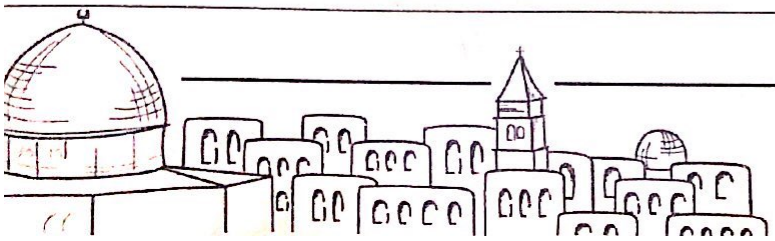
$X_1 = 28$

PC	
24	jal X_1, Average
28	

Average: ——

jalr rd, 0(X_1)

X_0 ←



Computed Jump

لما أكون بدي الـ PC اني أبدي اروح لـ

jalr X0, 1024(X0)

⟹ هاي اسمها Computed Jump

1024

Stack

Local Arrays or Structures

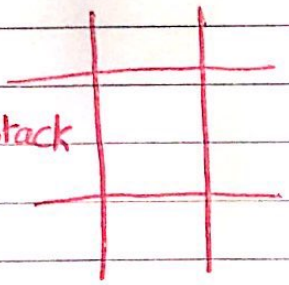
Spill registers

X_1 $(X_{10} - X_{17})$ ⟹ Non leaf

لما أكون بدي قيم راجوع

$(X_{20} - X_{27})$

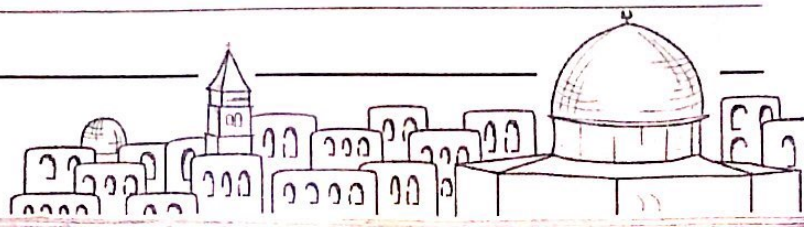
↳ saved register



بتبلش من most ال Address وبتنزل تنازلي

last in First out

8 Locations بنزل لتحت Stack pinters

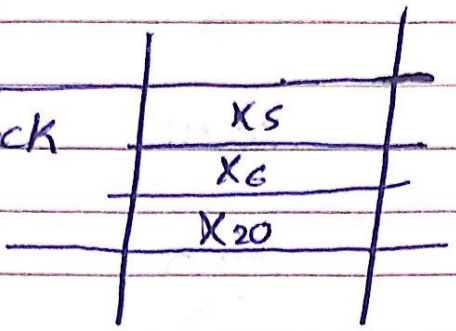


Leaf Procedure Example

- ما بنادي حالة
- ولا بنادي Func تانية

Example Slide 54

```
addi sp, sp, -8
sd x5, 0(sp)
addi sp, sp, -8
sd x6, 0(sp)
```



⇓

بتعد اعمل هيك
لبين ما يطلعني

more instructions

⇓

عشان ارجع الـ Stack

فاعمل بدل Load

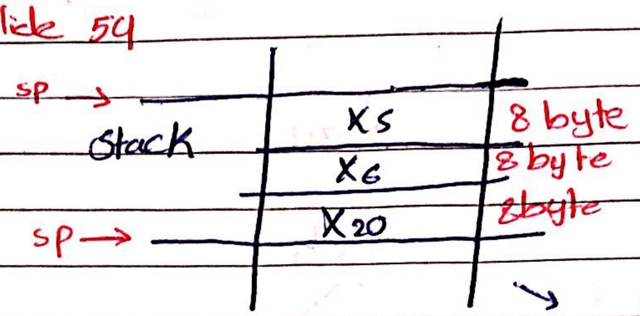
Leaf Procedure Example

- حابنادي داله
- ولا بنادي func ثاني

Example Slide 54

Save to the stack:

```
addi sp, sp, -8
sd   x5, 0(sp)
addi sp, sp, -8
sd   x6, 0(sp)
```



عاعمل ال load ال Data ماهترح لبس ال sp رح تتغير مكانه ، اعري الكاني رح أكتب فوقهم

↓

بقدر اعمل هيك بس رح يعطيني

عشان ارجع ال stack فاضي بعمل Load

more instructions

8 march 2020

Leaf Procedure Only store saved reg in the stack
We don't store the return address register (x_1) and the argument registers in the stack

```
add  x5, x10, x11
add  x6, x12, x13      → body of the Procedure
sub  x20, x5, x6
add  x10, x20, x0
ld   x5, 0(sp)
ld   x6, 8(sp)         → restore from stack
ld   x20, 16(sp)
addi sp, sp, +24
jalr x0, 0(x1)         → return
```

- Temporary register
  ما بحتاج أخزنهم بالStack وبقدر أكتب فوقهم

- Saved register
  لازم أخزنهم بال Stack وما بقدر أكتب فوقهم

Main:

0 ________
4 ________
8 ________
12 addi X_{10}, X_0, 5
16 addi X_{11}, X_0, 7 — z = leaf_example (5, 7, 3, 2) ← g h i j
20 addi X_{12}, X_0, 3
24 addi X_{13}, X_0, 2
28 jal X_1, Leaf_example
  ↓ (for X_1) 28 + 32
32 addi X_{17}, X_{10}, 0

Non leaf procedure

Caller need to save
return address
Any arguments and temp needed
After the call
← حسب ال func

$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$ $5 \times 4! = 5 \times 24 = 120$

$4! = 24$

$3! = 6$

$2! = 2$

$1! = 1$

$0! = 0$

Fact $(n)^{X_{10}}$

n * fact $(n-1)$ ↳ X_{10}

X_{10} function Call

```
20 fact: addi sp, sp, -16
   24    sd X1, 0(sp)
   28    sd X10, 8(sp)
   32    addi X5, X0, 1
If 36    bge X10, X5, Else
Part 40  addi X10, X10, -1
   44    addi sp, sp, 16
   48    jalr X0, 0(X1)
52 else: addi X10, X10, -1
   56    jal X1, fact
   60    addi X6, X10, 0
         ld X10, 8(sp)
         ld X1, 0(sp)
         addi sp, sp, 16
         mul X10, X6, X10
         jalr X0, 0(sp
```

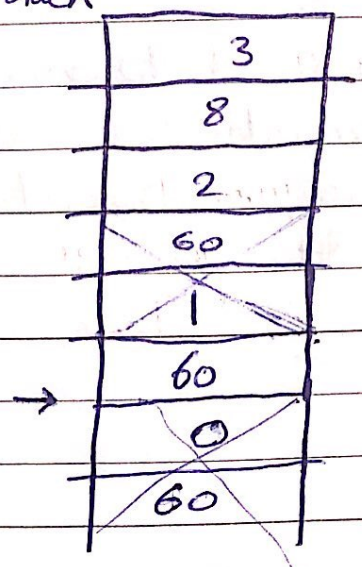
main

```
0 addi X10, X0, 3
4 jal , fact
```

$X_{10} = 3\ 2\ 1\ 0$

$X_1 = 8\ 66\ 66\ 60$

Stack



10 march 2020

long long int A[10];

⋮

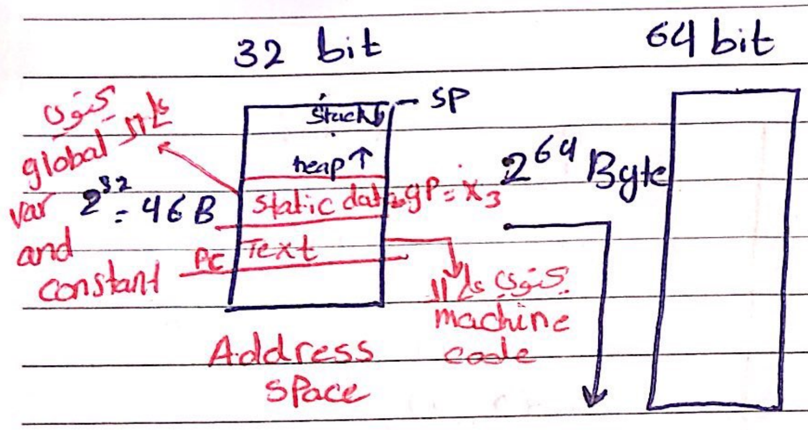
= A[7] + 9

↳ ___ (FP) x_8

FP - point to the first double word (Top of the stack)

ولها مكانه ثابت

* Memory layout



يكون موجود بالغي، وثابت

ld ___ (gp)

↓

12 bit

-2^{11} to $+(2^{11}-1)$

SP, fp 4 الـ procedure لكل (software)

الـ (hardware) بكون واحد

heap

لما نعمل structer ، الـ size مو fixed

بكونوا محجوزين بالـ heap

Function free → garbage

Slide 62:

```
Sum: addi  sp, sp, -8
sd   X1, 0(sp)
blt  X0, X10, if
  add  X12, X11, X0
  addi sp, sp, 8
  jalr X0, 0(X1)
if add  X11, X11, X10
  addi X10, X10, -1
  jal  X1, Sum
  Ld   X1, 0(sp)
  addi sp, sp, 8
  jalr X0, 0(X1)
```

(1) acc += n / (2) n = n - 1

Iteration

```
while(a>0)
{
b = b+ai
a = a-1;
}
b = 6
a = 0
```

recursion

a = 3
b = 0
c = Sum(a, b)

Sum(3, 0)
sum(2, 3)
sum(1, 5)
Sum(0, 6)

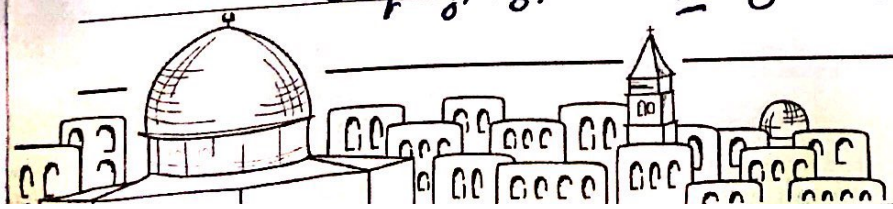
3+2+1 = 6

c = 6

Leaf Sum: blt X0, X10, L1

add X12, X11, X0
jalr X0, 0(X1)

L1: add X11, X11, X10
addi X10, X10, -1
beq X0, X0, Sum = jal X0, Sum



12 march 2020

⟹ character data

8 bit $2^8 = 256$ char

Ascii → 7 bit → $2^7 = 128$ character

long long int → 64 bits

int X = 1000 000 000;

↳ This takes in the memory 32 bits

= 1 ... '0' → "0011 0000"

8 bits

0 → 98

X0 30 → X0 39

Lwu → zero extended

Ld → lb, lh, Lw → word 4 byte

byte ↳ half word (16 bit / 2 byte)

Sd → Sb, Sh, Sw

Lx rd, offset (rs1)

double
word
halfword
byte

Sx rs2, offset(rs1)

d
w
h
b

offset + rs1

Word + rd

Sign extended يعني 32 64

Zero extended ← char السنتاكس اني اعمل لـ

Lwu X5, 32(X0)

Memory address = 32 = 32 + (X0)

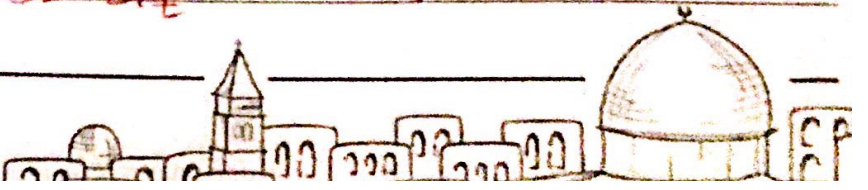
(X5) ← 32 bit Mem [32]

(X5) = 0000 ... 'Z' 'B' 'C' 'A'

32 bit

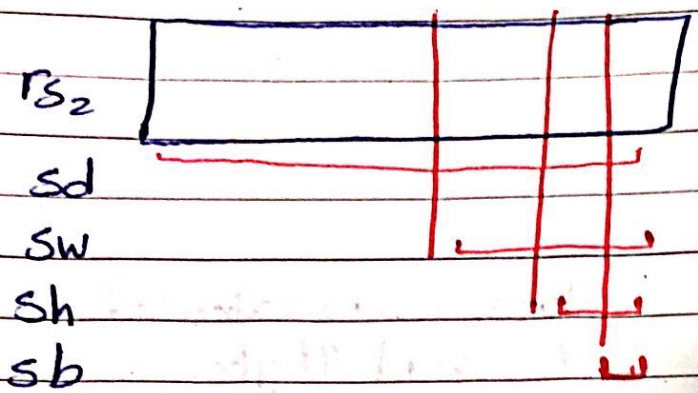
Zero extended

35	'Z'
34	'B'
33	'C' = 000 0011
32	'A' = 0100 0001



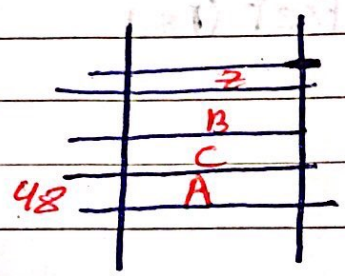
Store :

Sx rs_2, offset (rs_1)



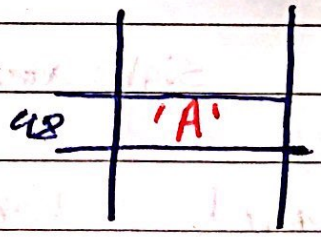
X_5 = 'Z', 'B', 'C', 'A'

Sw X_5, 48 (X_0)



Little Endian

Sb X_5, 48 (X_0)



* بحاجة اعمل extended لانو تخزين على الـ memory

Slide 67

```
i = -1
do
  i = i+1
  X[i] = y[i]
while (Y[i] != '\0')
```

→ lbu ... (X11)
; X size of data in byte
; X1 = i

```
Strcpy: addi sp, sp, -8
        sd   X19, 0(sp)
        add  X19, X0, X0
Loop    add  X5, X19, X11
X(5)←y[i] lbu X5, 0(X5)
        add  X6, X19, X10
X[i]←(X5) sb X5, 0, (X6)
        beq  X5, X0, Exit
        add  X19, X19, 1
        beq  X0, X0, Loop
Exit:   Ld   X19, 0 (sp)
        addi sp, sp, 8
        jalr X0, 0(X1)
```

addi rd, rs1, imm

imm ⇓ 12 bit → -2^{11} to $+(2^{11}-1)$ → 2047

Common case : Constant (12 bit)

Constant up to →

Load upper immediate (lui)

lui rd, imm

imm	rd	opcode
20	5bit	7bits

Clear bits [11:0]

rd

sign extended	imm	0---000
... b_{13}	b_{12}	b_{11} ... b_0

lui x_{20}, 0XFC23A

x_{20} =

11...1111111111	1111 : 0011 1010	00---0000
		b_{11} ... b_0

x_{20} = 0X FFFFFFFFC23A000

17 - 3 - 2020

معظم الحالات بستخدم ال 12 bits عشان يمثلوا ال constant

دايما مرات بنحتاج 32 bit

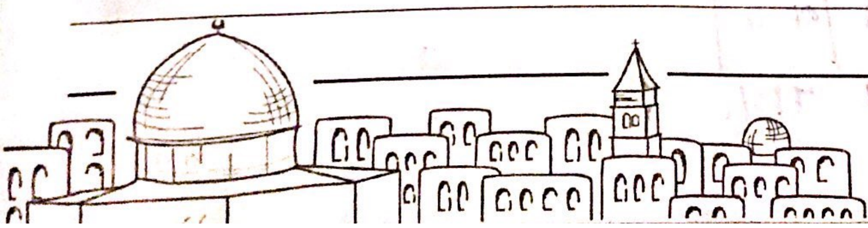
Load upper imm (lui)

lui rd, imm

imm	rd	opcode
20 bit	5 bits	7 bits

ex: lui x_{19}, 0X003D0 ✓

lui x_{19}, 976 →



(1)

sign extended ↑

X_{19} :

0X0000 0000	0X003D0	0X000
32 bits	20 bits	12 bits

0X
مش part من
ال number
بس عشان نميز ان
الرقم Hexadecimal

(2)

addi X_{19}, X_{19}, 0X500

X_{19} = 0X 0000 0000 003D0 000
0X 0000 0000 0 000 500
0101

X_{19} 0X0000 0000 00 3D0 500
The final answer

1111

Z = 0XF2D537AB

↓ X_{20}

Lui X_{20}, 0XF2D53
20 bits

addi X_{20}, X_{20}, 0X7AB

X_{20} = 0X FFFFFFFF | F2D53 000
0X 0000 0000 | 000007AB

X_{20} = 0X FFFFFFFF | F2D537AB

most sig bit is
1 بالعادة ما

12

$X_{19} = 0X003D0|800$

lui X_{19}, 0X003D0 → 0X003D1

addi X_{19}, X_{19}, 0X800

X_{19} = 0X 0000 0000 003D0 | 000

0X FFFF FFFF FFFFF | 800

0X 00000000003CF | 800

مو نفس الجواب

Hexadecimal

C 10

3D0

1

3CF

$(10)_{10} = (16)_2$

2's complement
$(11111111)_2 = (-1)_{10}$

0X000 000 000 3D0 800

X_{20} = 0X 3D2C7 D57
20 bits | 12 bit

1101

sign extended

لما اجمعها 2 اعدل

رح يطلع 1 رقم غلط

lui X_{20}, 0X3B2C8

addi X_{20}, X_{20}, 0XD57

X_{20} = 0X 0000 0000 3B2C7D57

اصوله

Risc-V

$X_{21} = 0X9A5CFA38$

A → 1010 → بلزم اجمع 1

lui X_{21}, 0X9A5D0

addi X_{21}, X_{21}, 0XA38

?

X_{21} = 0XFFFFFFFF9A5D0	000
FFFFFFFFFFFF	A38
X'_{21} = 0XFFFFFFFF9A5CF	A38

بطرح واحد

9A5CF + 1 = 9A5D0

beq rs_1, rs_2, label

label → Target

SB format

imm[11]

الترقيم مبلش من 1 - 12 bits

imm[12]	imm[10:5]	rs_2	rs_1	funct3	imm[4:1]	imm[11]	opcode
	7	5	5	3	5		7

لازم تكون صفر عشان يسهل عليه يجيب

sign extended

label ≡ 12-bits immediate

imm [11:0] → store

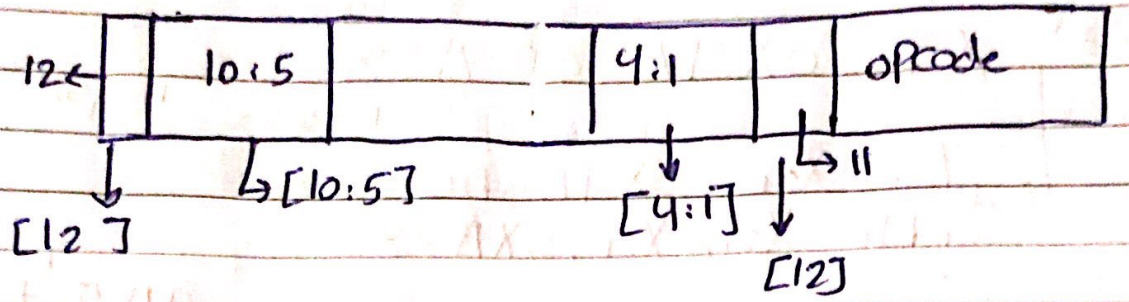
imm [12:0] → branch

ليش هيك موزعين الأرقام؟

عشان يخلوا ال bits تبعت متشابهين أكبر قدر ممكن مع ال sd

sd	imm [11:5]		rs_2	rs_1	funct3	imm [4:0]		opcode
beq	[12]	10:5	✓	✓	✓	[4:1]	[11]	✓

imm[12:1]



forward ← beq X_{10}, X_{12}, Exit

12 bits

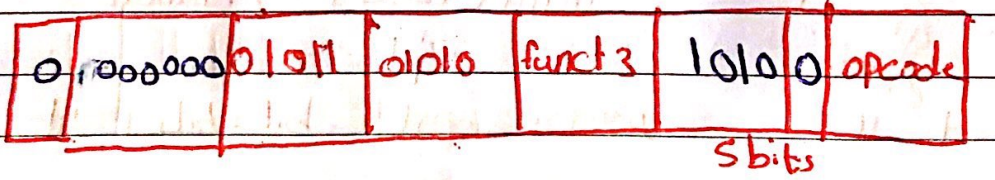
Exit

Target address = PC + 2 X imm
of
branch

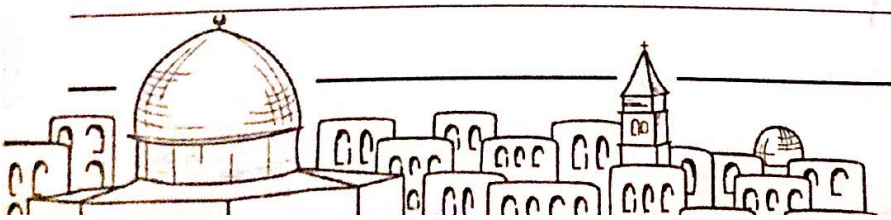
PC
0 beq X_{10}, X_{11}, Exit
4
8
12
16
Exit 20...

$20 = 0 + 2 \times imm$

$(10)_{10} = (0\text{-}000101\,0)_{2}$ imm $= \frac{20}{2} = 10$



اول اشي بجلو بقارن X_{10} و X_{0}
لو متساويين برجع عند ال exit



Loop 0 ___
4 ___
8 ___
12 ___
16 ___
20 ___
24 beq ___, ___, Loop

Target
6 instructions
6 X 2 = -12
سالب لانه
بطلع لفوق

instructions كمية
من دون
beq
Target لحد ال

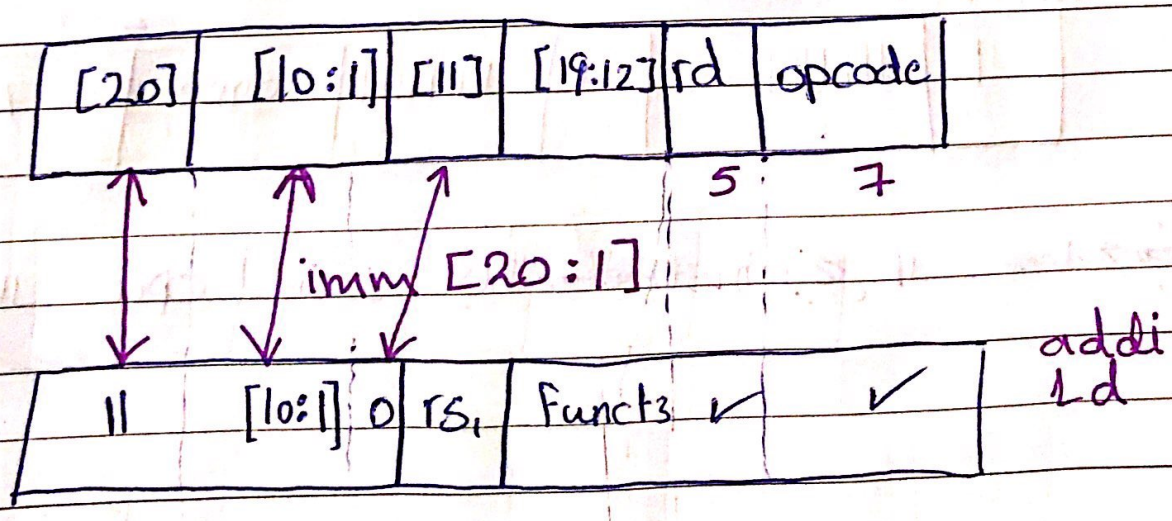
Target = PC of branch + 2 X imm

0 = 24 + 2 X imm

imm = $\frac{-24}{2}$ = -12

19 march 2020

jal rd, label



jal X_1, fact

Target = pc + 2 X imm
of
jal

0 = 16 + 2 X Loop
Loop = -8

Loop 0 —
4 —
8 —
12 —
16 jal X_0, Loop

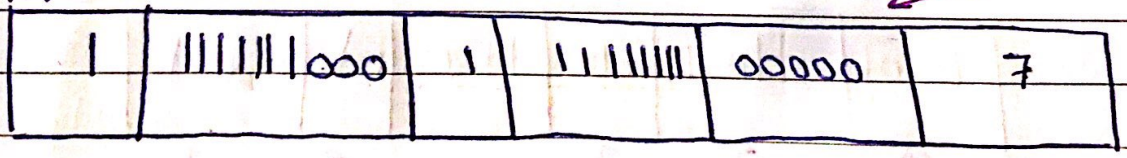
backward ← Target لما يكون ال
سالب ← imm يكون ال

1 — 1 instruction → 2 half word
2 X 4 = 8 → -8

Target = 16 + 2 X -8
16 - 16 = 0

0 —
4 —
8 —
12
16 jal X_0, 8

$(-8)_{10}$ = (
+8 = = ...0001000
-8 ...11111111000



ال cpu لما ينفذ هاي ال instruction بتنقص ال imm

branch (SB)

	rs₂	rs₁	funct3		opcode
7	5	5	3	5	7

imm [12:1]

UJ

[20]	[10:11]	[11]	[19:12]	rd	op
20				5	7

imm [20:1]

Target = PC + 2 X imm

PC - relative addressing

ليش بلشنا من index ① ؟ الـ imm قزت بوحدة لـ half word

shift left by 1

فـ لما احسب الـ Target بضرب بـ 2 دايمًا

PC

$\rightarrow$ beq X_{10}, X_{12}, label

↓

60

60 Label

X → Absolute Addressing

	beq	jal
Label	12 bit	20 - bit

± 2048 byte

ما بصير يكون اكبر ... بس هاي الحدود وهاد الاشي قليل عشان هيك ما بستخدم الـ Absolute

PC- Relative Address

beq	jal	
12 bit	20 bit	
±2048 half word	± 1M half word	

beq ↑ ↓ 2^{19} Label

PC = 2^{64} byte
64 bit

most offset branch ± 2K half word → ± 4K Bytes
most offset jal 2^{19} ± 512 K hW → ± 1M Bytes

80000 Loop: Slli
80004 ___
80008 ___
80012 bne X9, X29, Exit (6)
80016 ___
80020 beq X0, X0, Loop (-10)
80024 Exit

if X9 and X29 not equal?
80012 + 2X6
= 80024
↓
Exit

Exit = $\frac{80024 - 80012}{2} = (6)_{10} = (0\cdots000110)_2$

Target beq?
80020 + 2X-10
= 80000
↓
Loop

Target = PC + 2 X imm

Loop = -5 X 2 = -10 = $(-10)_{10}$ = $(1\cdots 1110110)_2$
instructions

80000 = 80020 + 2 X Loop

$\frac{-20}{2} = -10$

12	10:5	rs_2	rs_1	4:1 , 11	op

beq rs_1, rs_2, L1 ⇒ bnq rs_1, rs_2, L2
Jal x_0, L1

L2: ——

±4K byte

±1MB

Range الى خارج L1:

L1 ⟶ 1MB اكبر من

↓

bne rs_1, rs_2, L2
jal x_0, L1
L2:

1MB

jal x_0, L1

L1

jal x1, Target

↓

20 bit

±1MB

64 bit value

Address

Target

lui x10, Imm

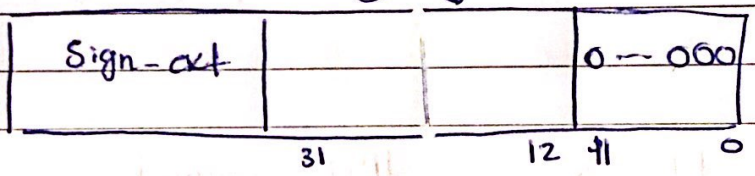
jalr x1, (x10)

12 bit

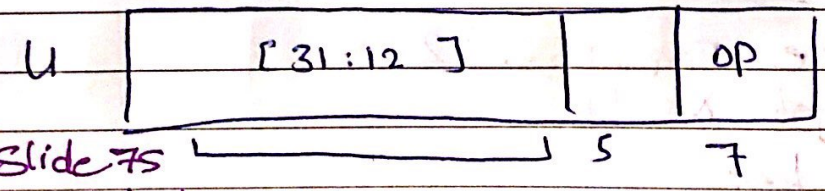
↓

32 bit

lui rd, imm



20 bit



Slide 75

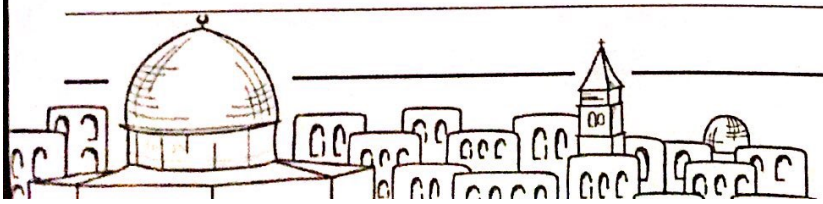
jalr rd, offset(rs1)

absolute Address

beq → PC relative

jal → PC relative

PC = Sign ext(offset) + (rs1)



± 2048

0 jalr X1, 100 (X0)

100 + (X0)

Pc = 100

Pc → 0X 0000 0000 0000 0000

0 jal X0, L1

±1MB

Lui X19, 0X 3ABC1

addi X9, X9, 0X2F5

jalr X0, 0X2F5 (X19)

L1:

0X 0000 0000 3ABC 12F5

20 bit | 12 bit

32 bit

Pc = (X19) + 0X2F5

2.1 - march 2020

Appendix A.5

0001 (+1)

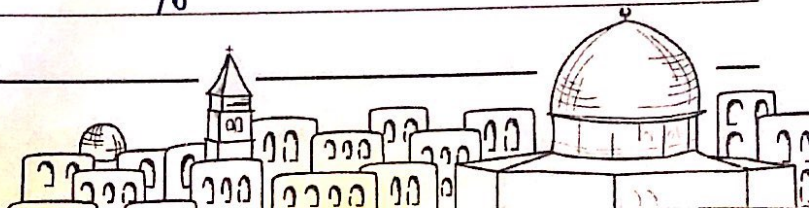
↓↓ negation (1)

1111 (-1)

+ 1110

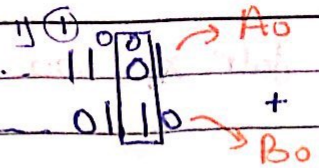
1 (2)

$(1111)_2 = (-1)_{10}$



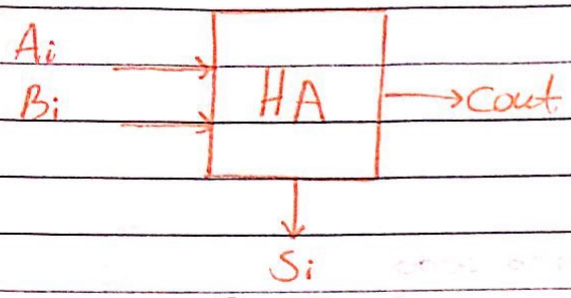
A = A_{63}

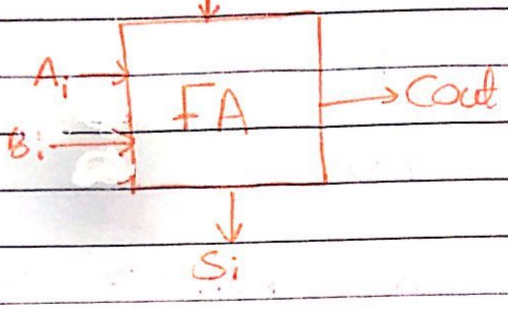
B = A_{63}

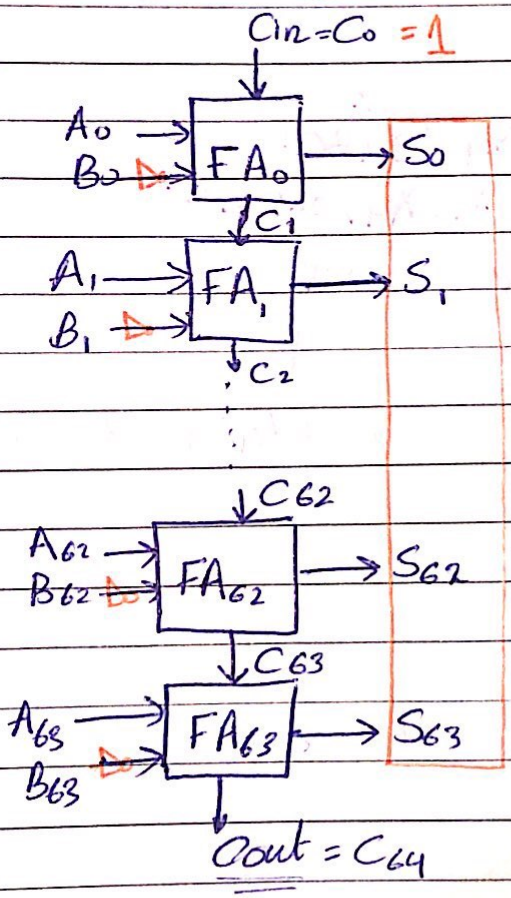


0011

Cin = Carry out





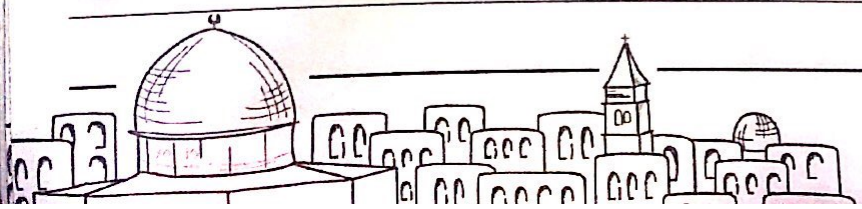


64-bit RCA
Ripple carry Adder

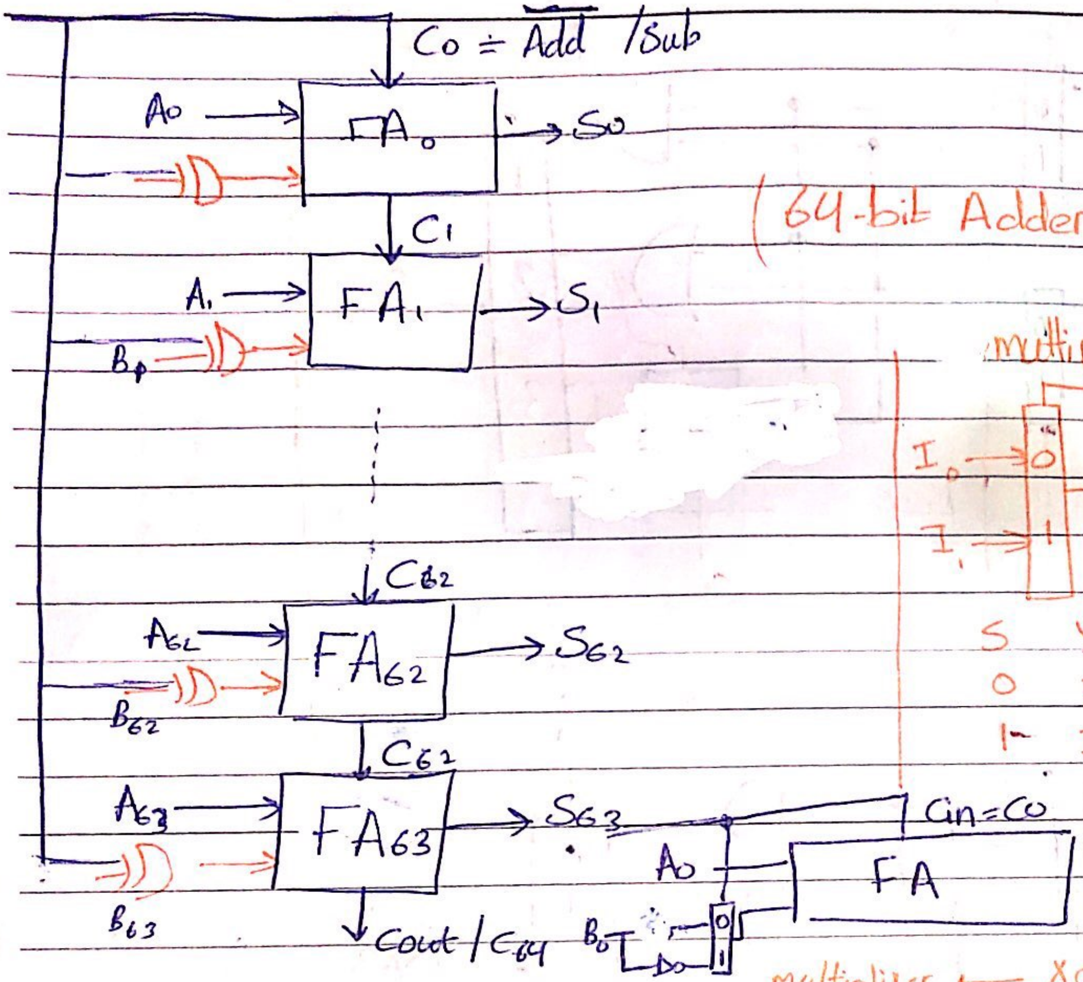
$A-B=A+(-B)$

$=A+\overline{B}+1$

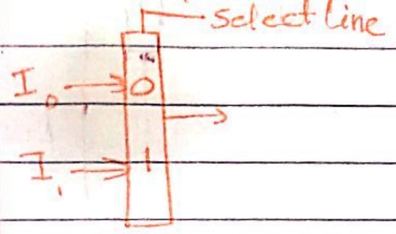
64-bit RCS



(64-bit Adder/Subtracter)



multiplixer



S	y
0	I_0
1	I_1

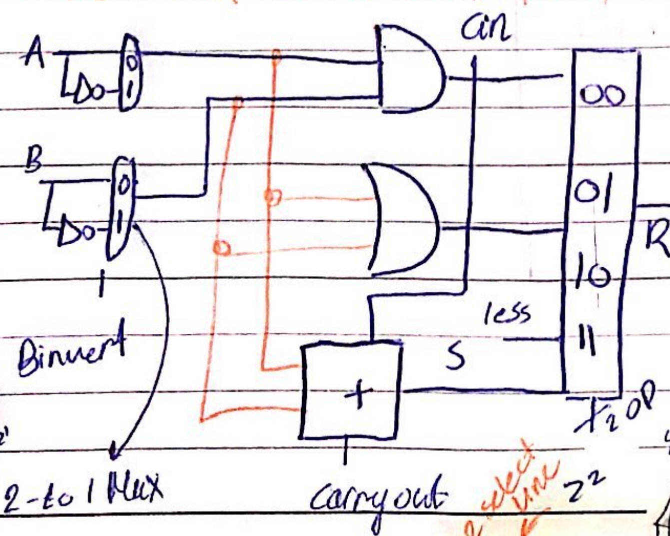
multiplixer ← Xor ا سقبل ال

and
or
Xor

$A = A_{63}$ ... A_0 parallel العملية بتصير

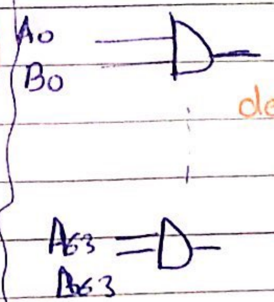
$B = B_{63}$ ... B_0 bitwise operation من العملية

1bit Alu Slice (and/or/add/sub)



1 select line

2 select line



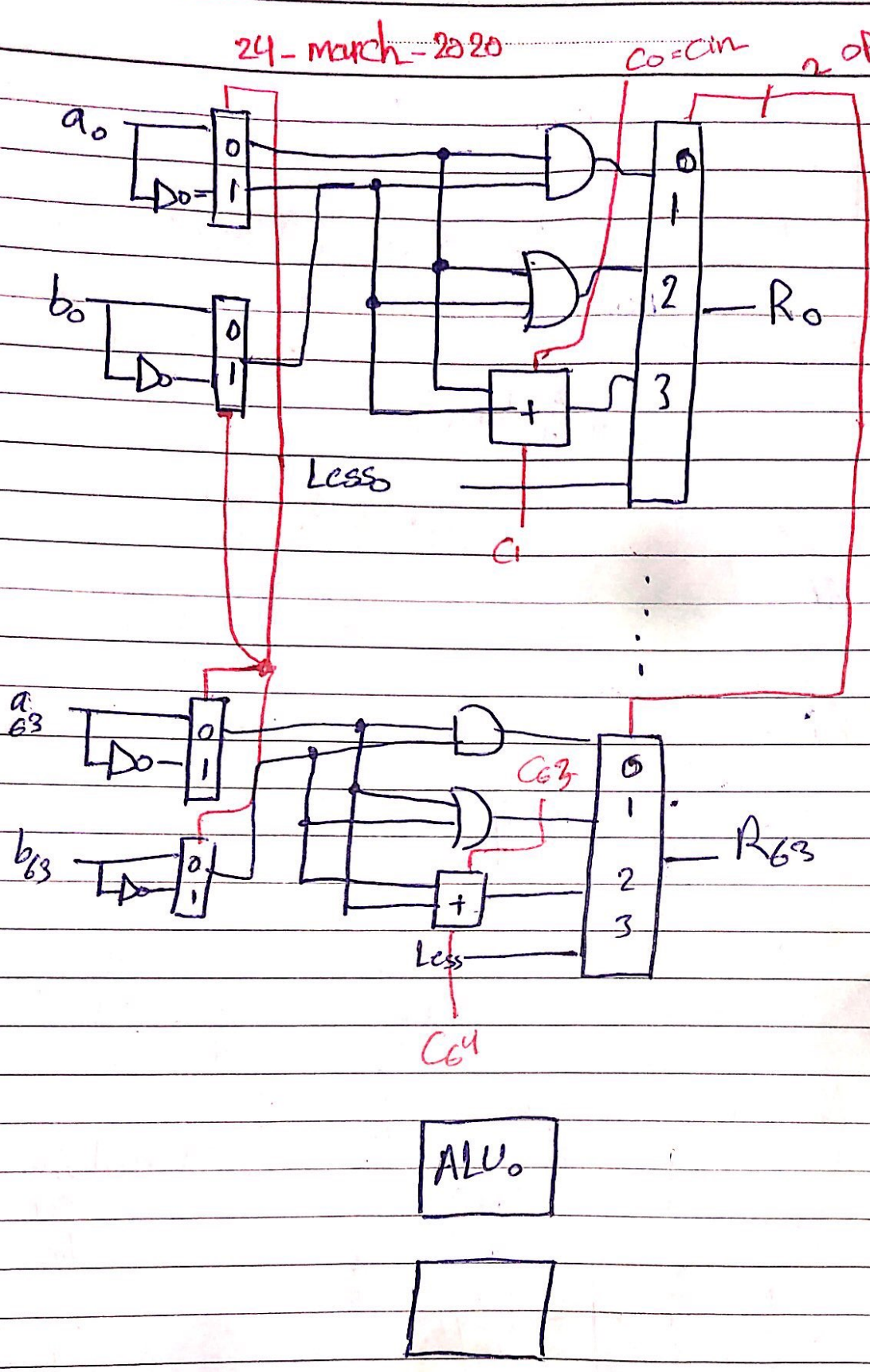
رح يتكرر الdesign 64 Times

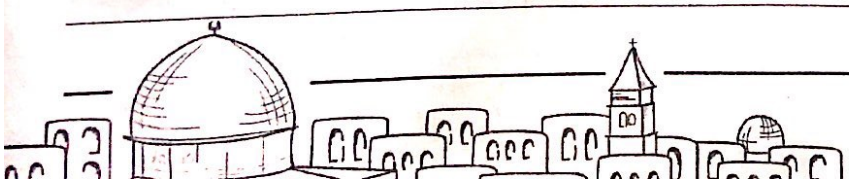
ال And or ← Parallel

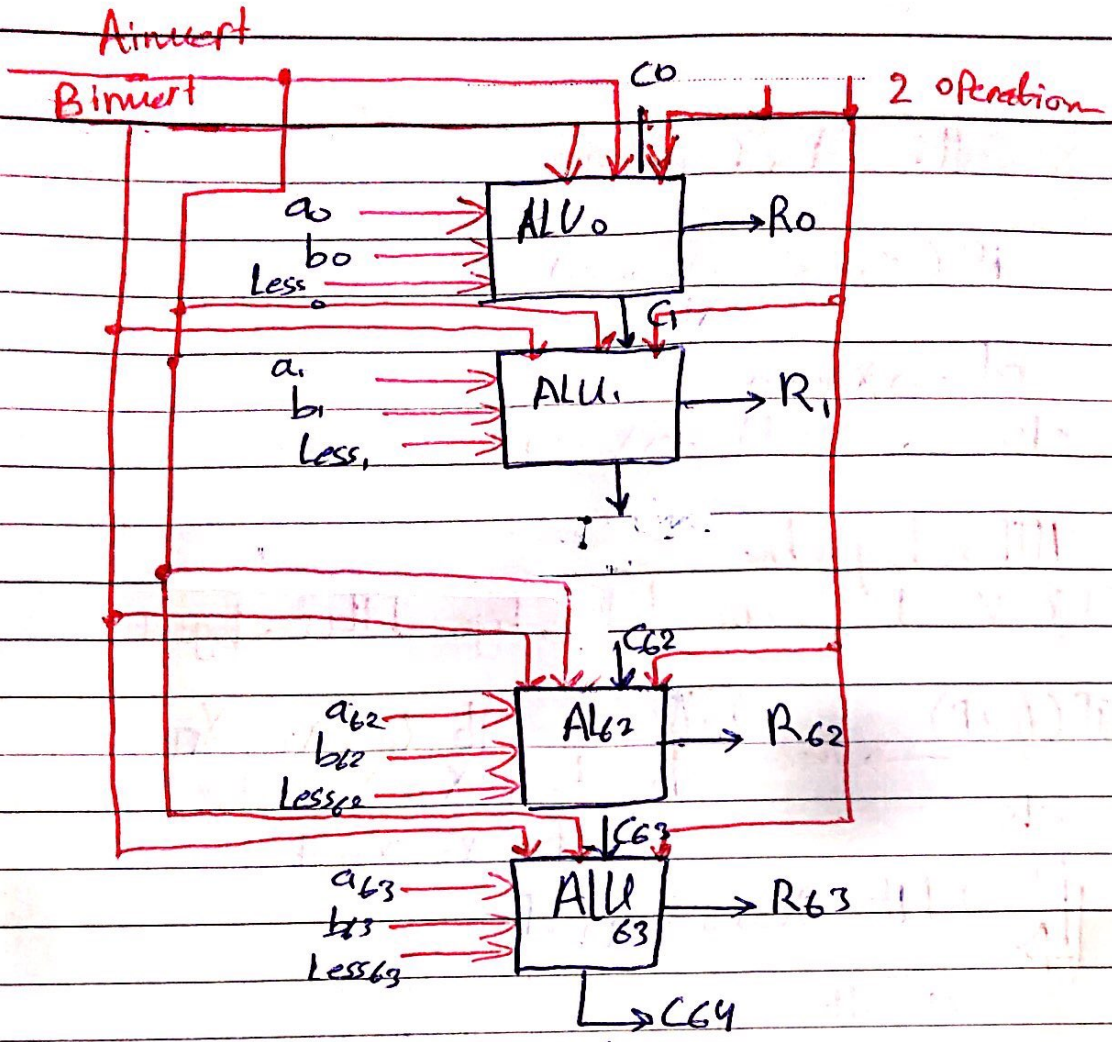
ال FA مش bitwise

بتعتمد على ال block الي قبلها

24-march-2020



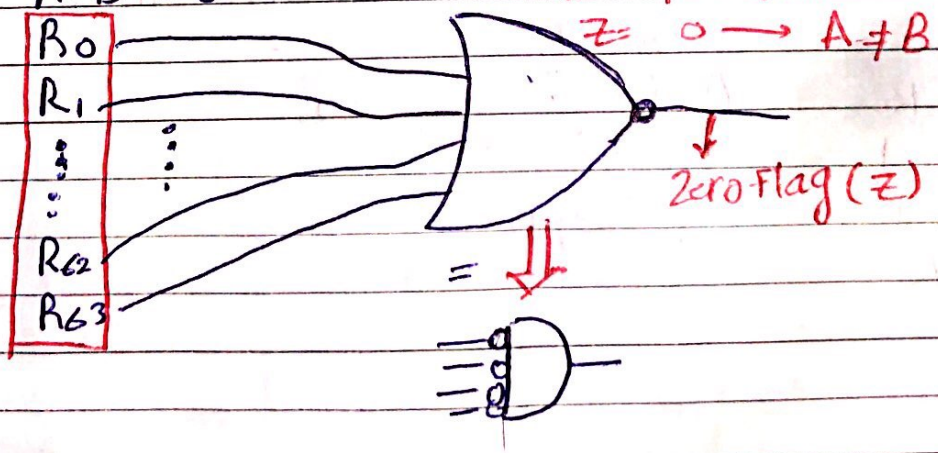




beq rs_1, rs_2, Label

$(rs_1) - (rs_2)$

A - B = 0



Set-if-less-than

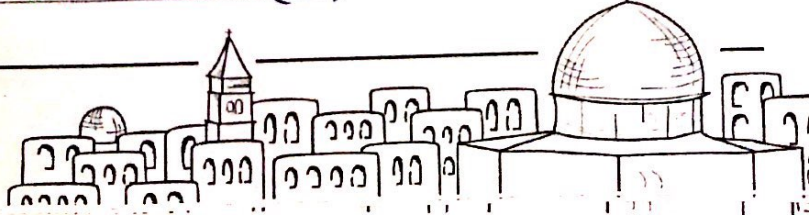
slt rd, rs_1, rs_1

slti rd, rs_1, imm

if $(rs_1) < (rs_2)$

rd = 0x0000...0001

else (rd) = 0x0000...0000



slti rd, rs1, imm

if (rs1) < Sign ext (imm)

rd = 0x0000 0001

else → (rd) = 0x0000 — 0000

HIPS: beq, bne

RISC-V: beq, bne, blt, bge, bltu, bgeu

if (A > B)

X_{19} ← — → X_{20}

else

HIPS ⇒ slt X_5, X_{20}, X_{19}

beq X_5, X_0, Else

↳ bne X_5, X_0, If

RISC-V ⇒ (1) blt X_{20}, X_{19}, IF

(2) slt

beq

slt / slti

(rs1) < (rs2) / (rs1) < imm

A < B

A − B : [ ] 64 bit

most Sig

1 → negative

0 → positive - zero

negative

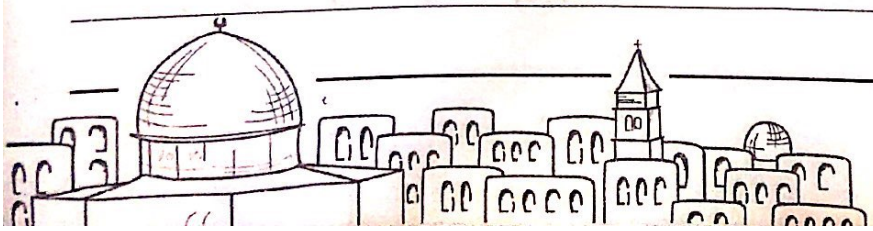
Rs — R0

00000000 — 1

Zero - Positive

Rs — R0

00000000



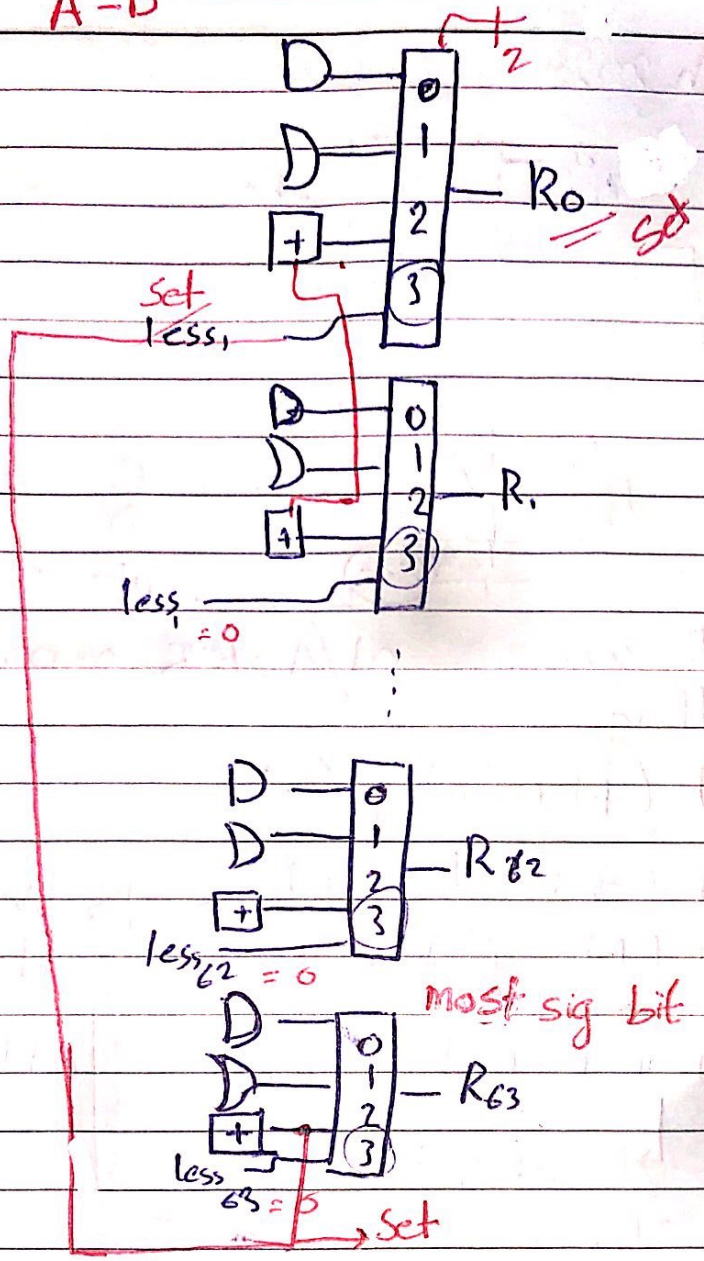
A - B

Negative

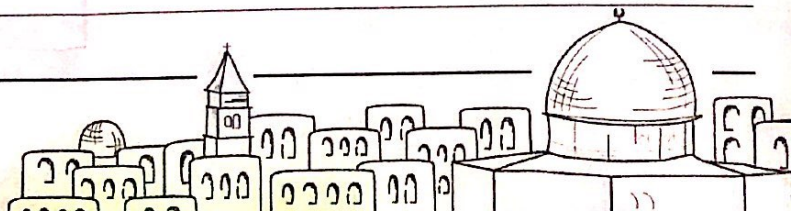
0000 --- 1

else

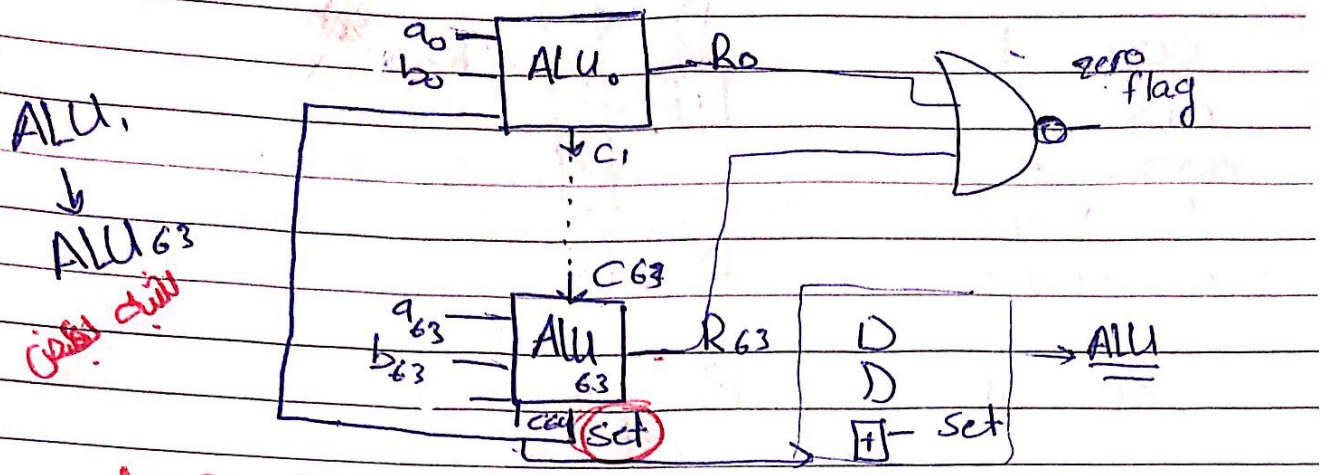
0000 ... 0



	Ainvert	Binvert	Co	op(2)	Binvert = Co
add	0	0	0	10	نفس القيمة
sub	0	1	1	10	↓
and	0	0	X	00	Bnegate
or	0	0	X	01	↓
beq	0	1	1	10	Binvert+Co
slt	0	1	1	11	



26 March 2020



A<B : R = 00000 — 01 / A ≥ B : R = 00000 — 00

over flow

(1) (+) + (+) = (−)

(2) (−) + (−) = (+) ⟹ over flow

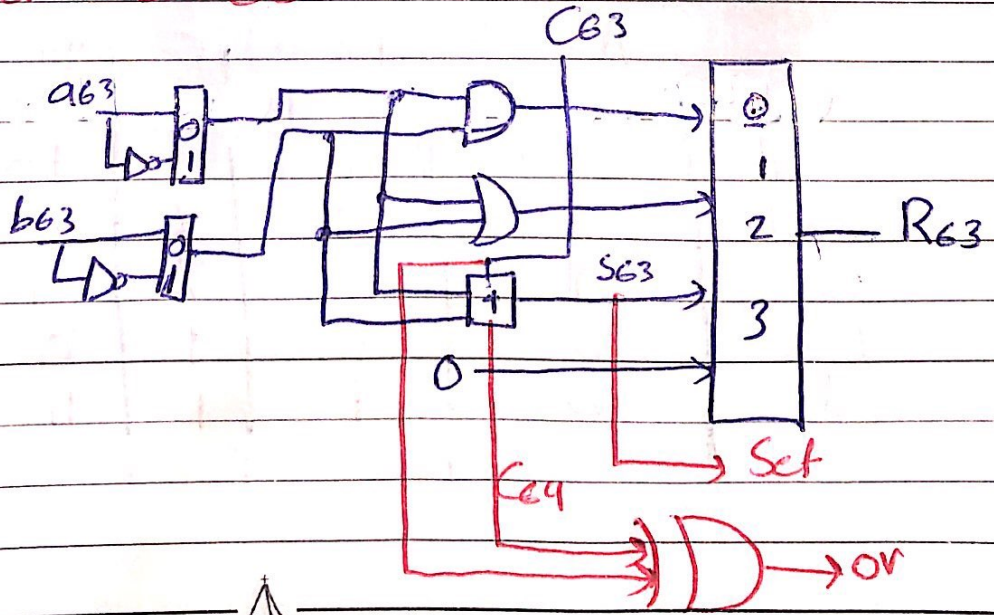
(3) (+) − (−) = (−)

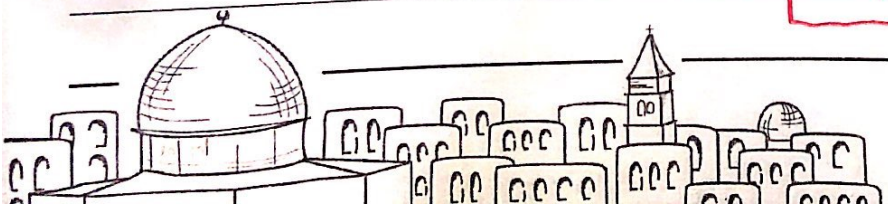
(4) (−) − (+) = (+)

↳ (−) + (−) =

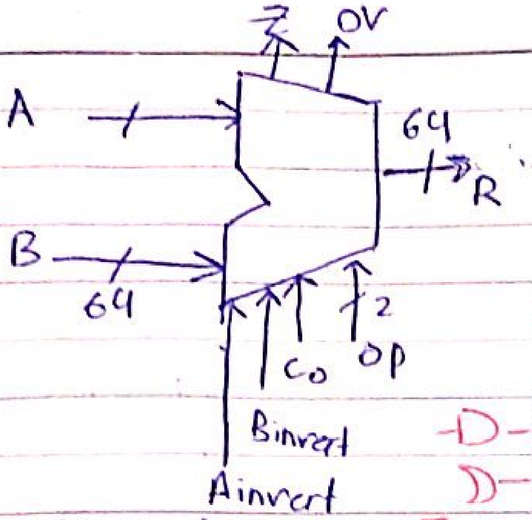
Add A+B
Sub A−B = A+B+1

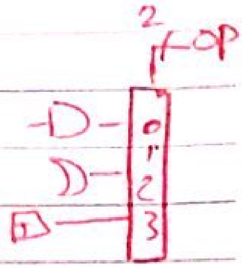
ALU Slice 63





and - or - add - sub
beq - slt - Nor

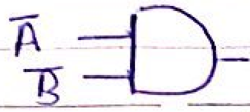




nor rd, rs1, rs1
(rd) ← (rs1) nor (rs2)

rd ← (not(rs1)) And (not (rs2))

↓
هاد بحتاج الحل
Design الـ



Nor operation

Slt
A−B < 0 ⇒ A < B → 00000 --- 01
A−B ≥ 0 ⇒ A ≥ B → 00000 --- 00

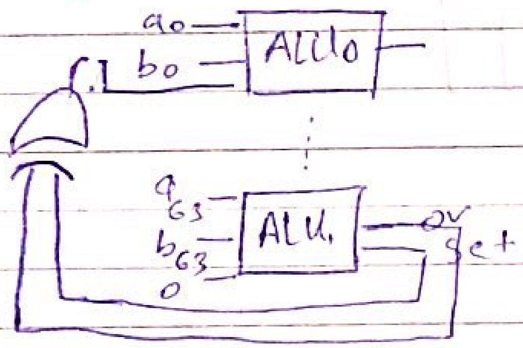
lesso = Set_{S63} → من طلعنا الديزاين

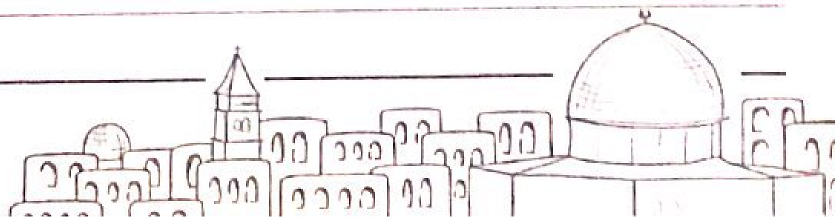
+ 1 ... 1 ... ⓪	⇓ overflow	+ 0 ... 0 ... ①

تعكس الاشارة صح less

Xor function

set	ov	lesso
0	0	0
0	1	1
1	0	1
1	1	0

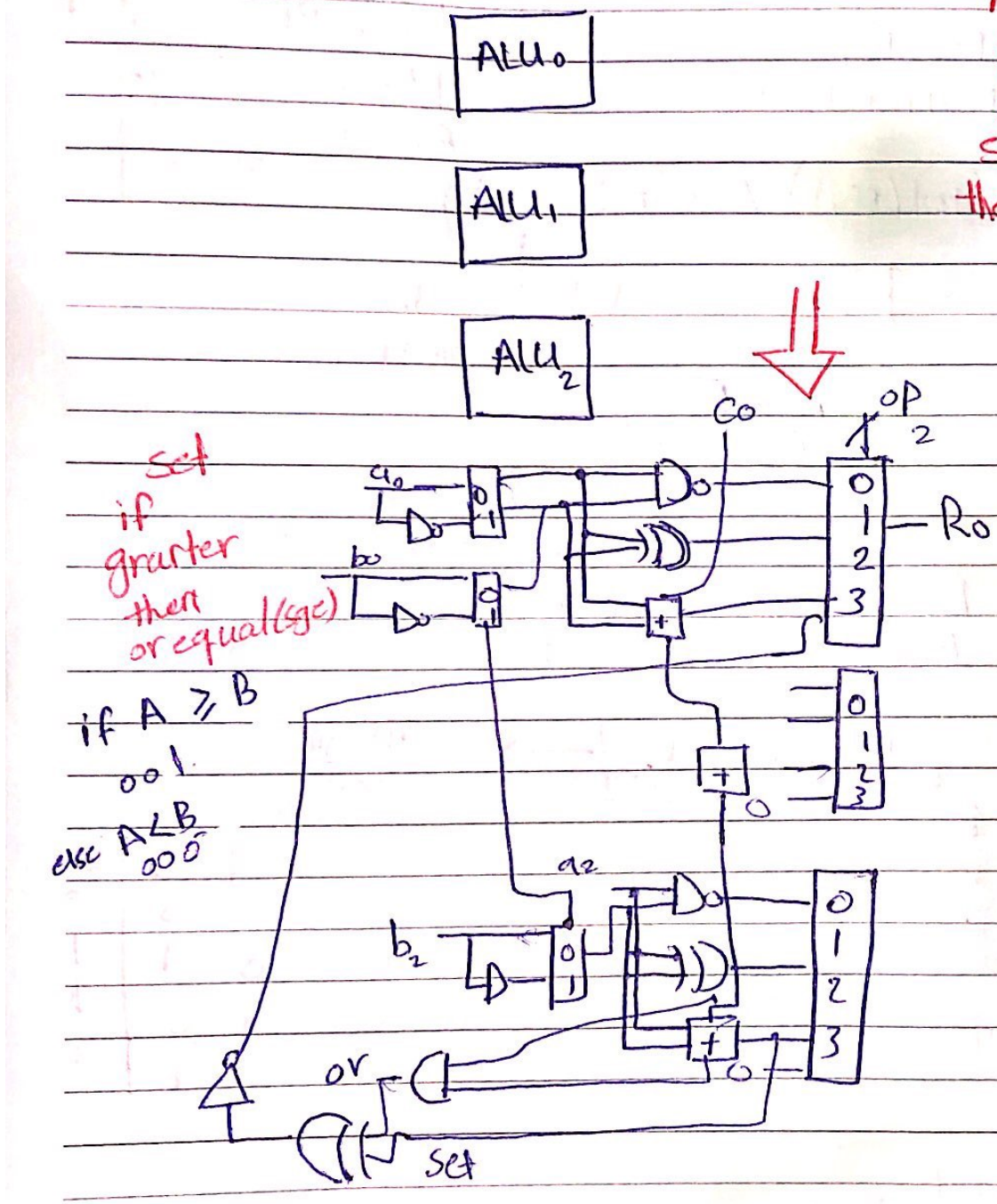


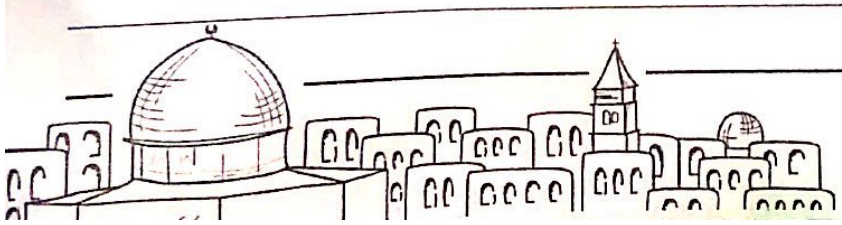


سؤال الامتحان

3bit ALU

- A Nand B op=00
  A Xor B op=01
  A+B :op= to set if grater than orequal

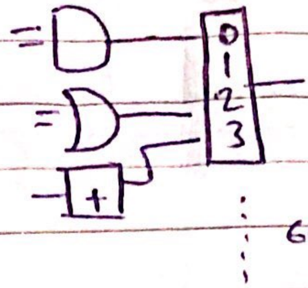




29 - march - 2020

Appendix A.6

And/or ال



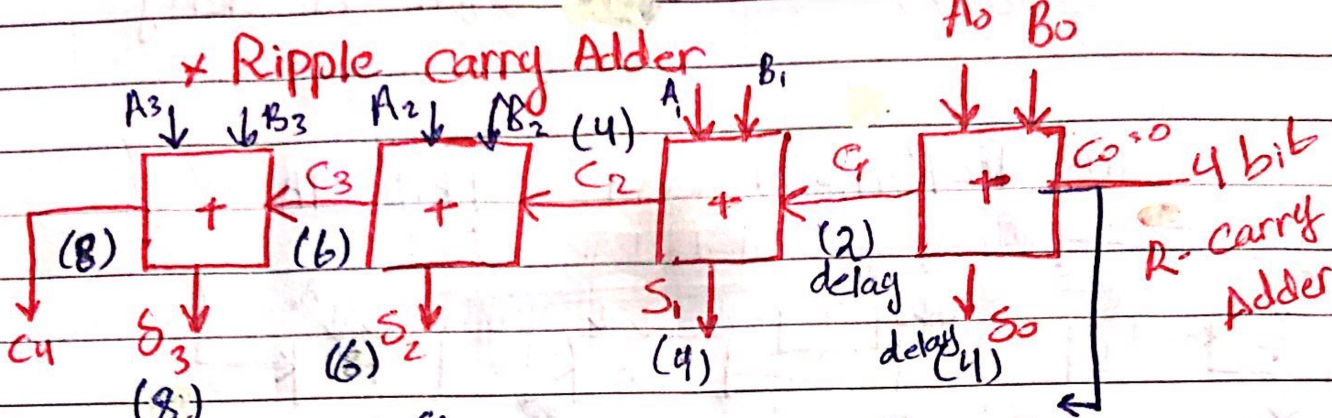
ما بنحتاج نعمل نشتغلوا ← شانمهاك ترتيبه

in Parallel

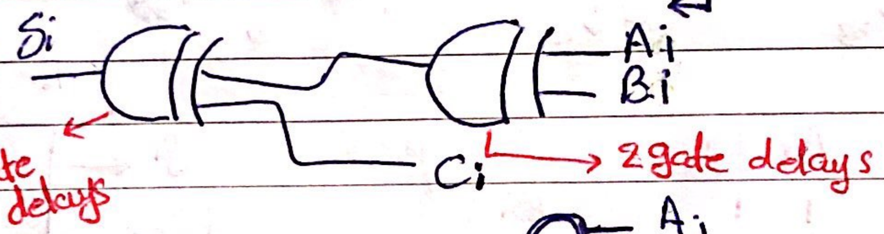
* البطأ

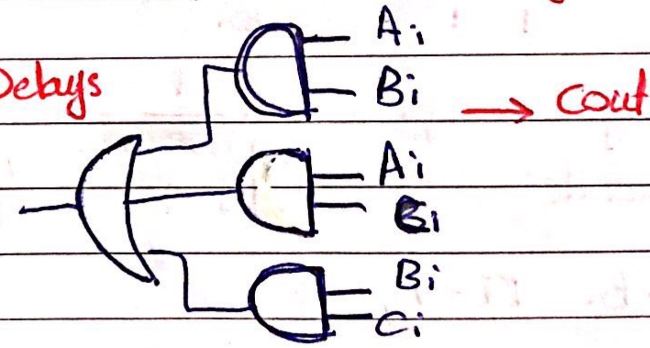
سؤال Sub/Add

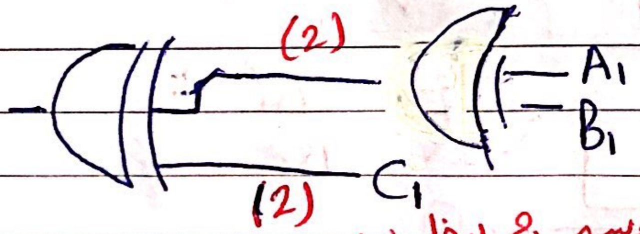
* Ripple carry Adder



4 gates Delays

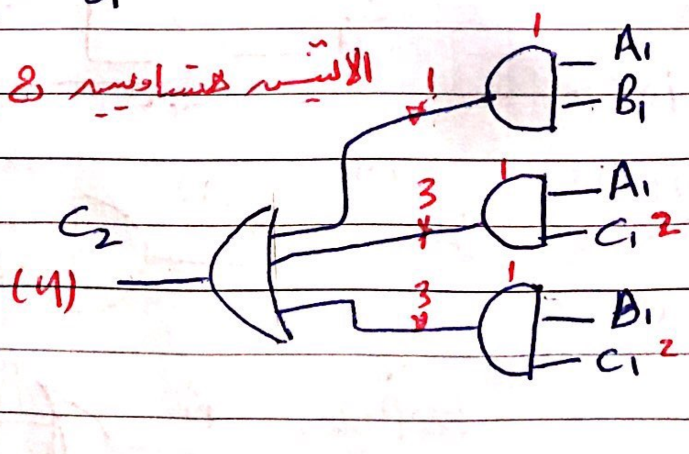






(2) ← Xor   دار (2) يدخل 2 الإثنين متساويين

2 + 2 = (4)



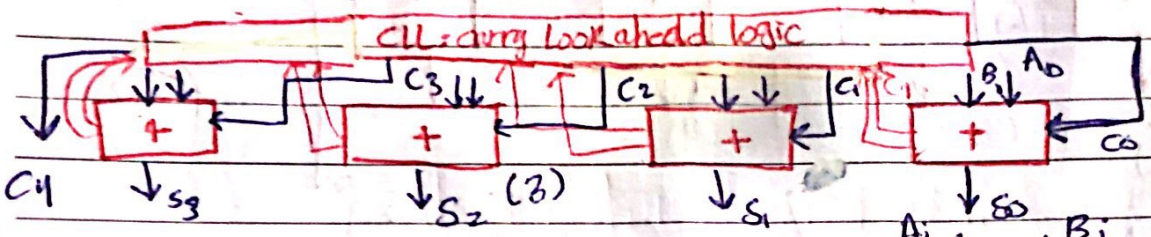
Delay of 4-bit RCA = 8 gates delay

The worst case → 8 gate delay

(Slow)

* Delay of n-bit RCA = 2×n gate delay

Carry lookahead adder 4 bit CLA



Ai Bi

1 1

Carry generate

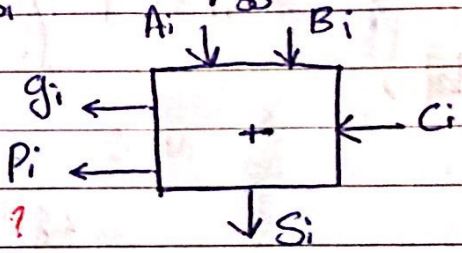
Ci or 1

Bi 1

Ai 1

1 0

1 1

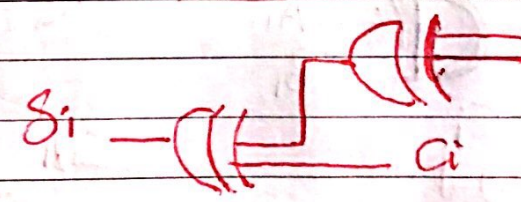


Ci ?

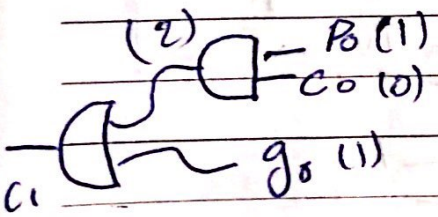
Ai 1

Bi 0

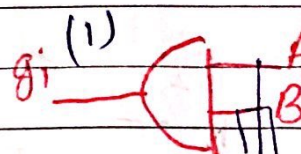
$g_i = A_i \cdot B_i \quad P_i = A_i + B_i$



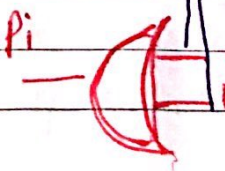
$C_1 = g_0 + P_0 C_0$



carry generate



carry Propagate

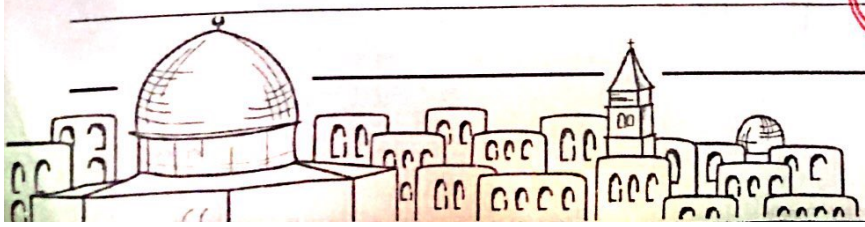


$C_1 = g_0 + P_0 C_0$

$C_2 = g_1 + P_1 C_1$

$= g_1 + P_1 (g_0 + P_0 C_0)$

$= g_1 + P_1 g_0 + P_1 P_0 C_0$



$C_3 = g_2 + P_2 g_1 + P_2 P_1 g_0 + P_2 P_1 P_0 C_0$

$C_3 = g_2 + P_2 C_2$

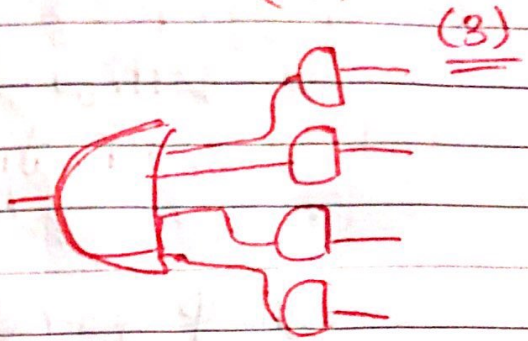
SOP (2 level Circuit)

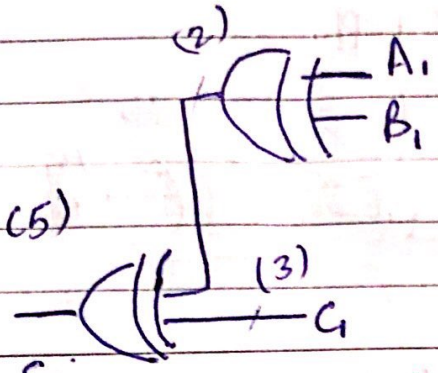
↳ Sum of Product

$C_4 = g_3 + P_3 g_2 + P_3 P_2 g_1 + P_3 P_2 P_1 g_0 + P_3 P_2 P_1 P_0 C_0$

(SOP)

S_0

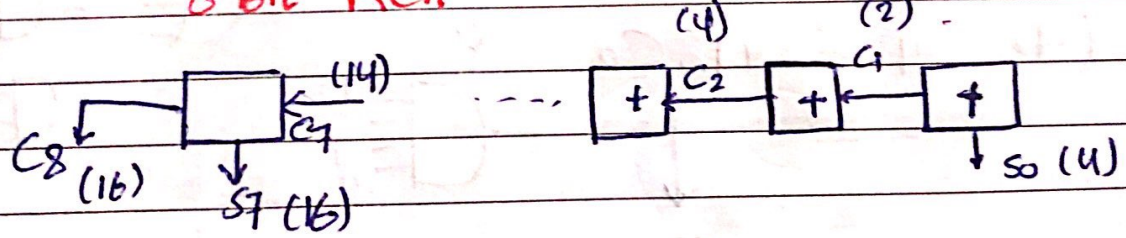




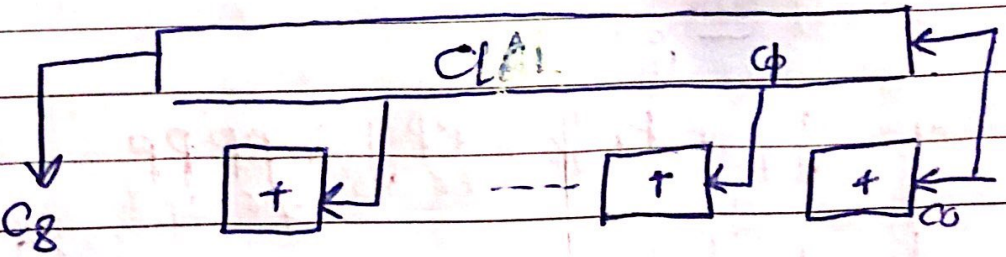
Delay of 4 bit CLA = 5 gate Delay

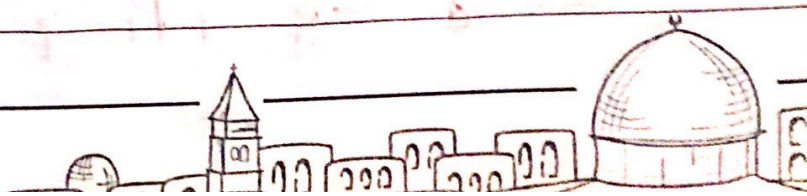
31 March 2020

8 bit RCA

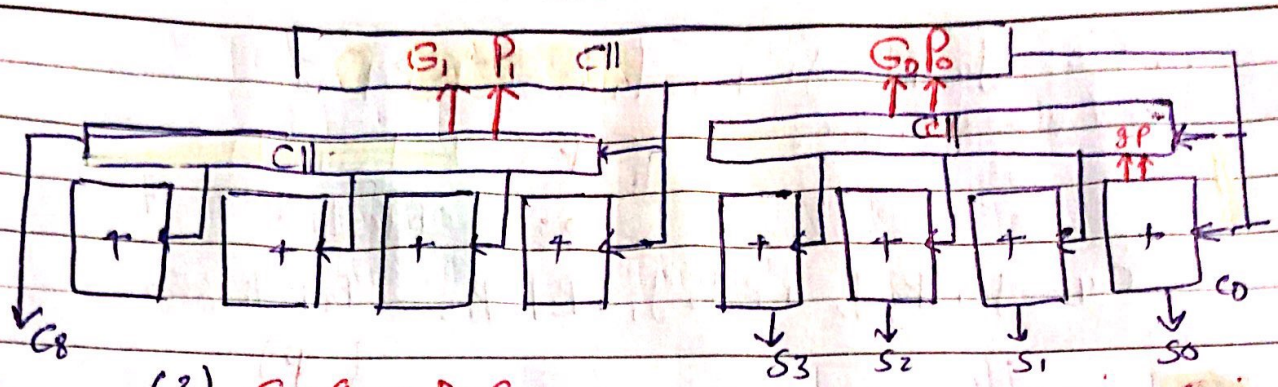


8 bit CLA





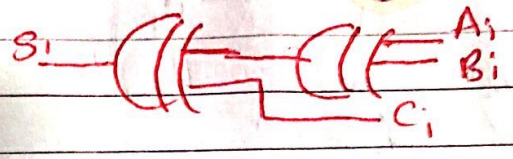
8bit CLA



(3) $C_1 = g_0 + P_0 C_0$

(3) $C_2 = g_2 + P_1 g_0 + P_1 P_0 C_0$

(3) $C_3 = g_2 + P_2 g_1 + P_2 P_1 g_0 + P_2 P_1 P_0 C_0$

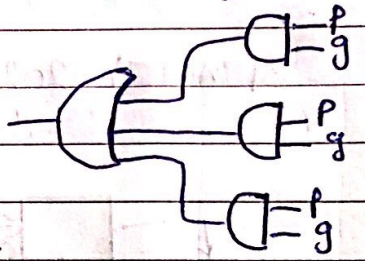


G = group generate

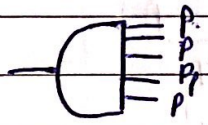
delay | P = group Propagate

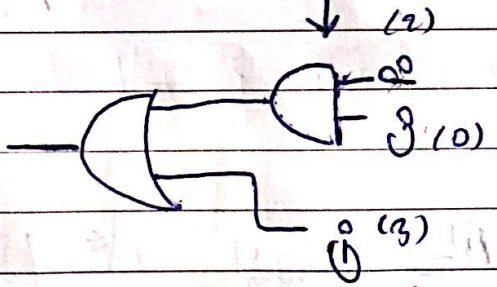
(3) $G_0 = g_3 + P_3 g_2 + P_3 P_2 g_1 + P_3 P_2 P_1 g_0$

(2) $P_0 = P_3 P_2 P_1 P_0$



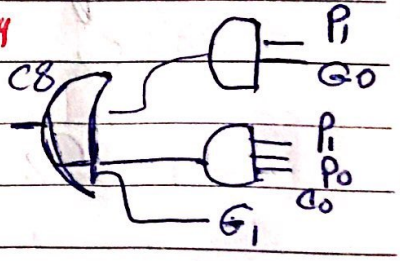
(4) $C_4 = G_0 + P_0 C_0$



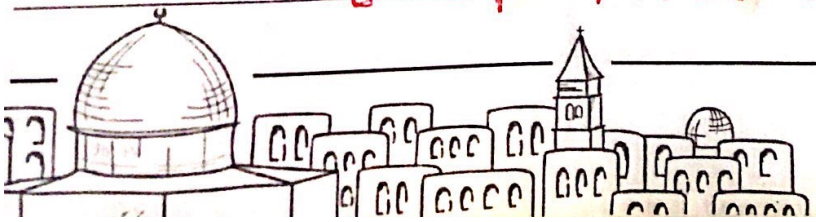


(3) $G_1 = g_7 + P_7 g_6 + P_{76} g_5 + P_{765} P_4$

(2) $P_1 = P_{7654}$



$C_8 = G_1 + P_1 C_0 + P_1 P_0 C_0$



(6) $C_5 = g_4 + P_4 C_4$ ← جاي

(6) $C_6 = g_5 + P_5 g_4 + P_5 P_4 C_4$

(6) $C_7 = g_6 + P_6 g_5 + P_6 P_5 g_4 + P_6 P_5 P_4 C_4$

بطلب مني احسب القيم ← $a_7 \ldots a_0$ قيم → بالامتحان

$b_7 \ldots b_0$

Delay أو ال

Delay of 8-bit CLA : 8 gate delay

Delay of 8 bit RCA = 16 gate delay

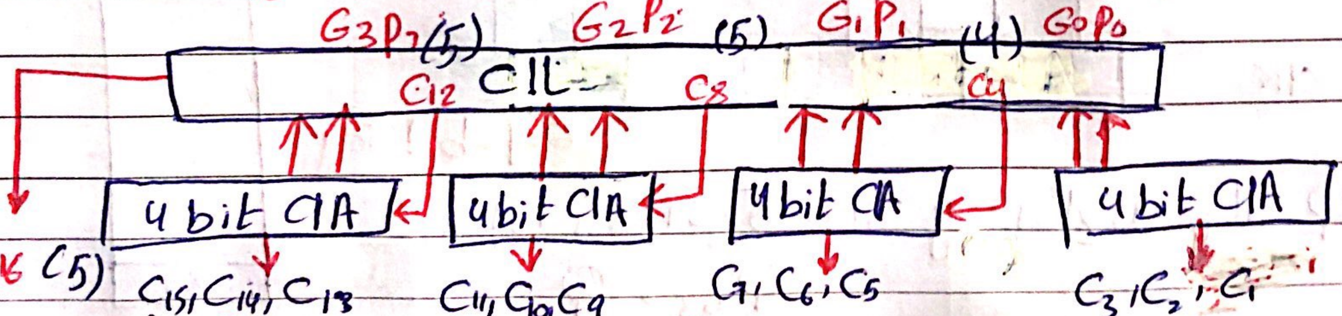
$C_8 = G_1 + P_1 G_0 + P_1 P_0 C_0$

3 input → max

more Area And Power

* 16 bit RCA → 32 gate delay

16 bit CLA



(5) $C_{12} = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0$

(5) $C_{16} = G_3 + P_3 G_3 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 G_0$

(7) S_{12}	(7) S_8	(6) S_4	(4) ← S_0
$S_{15} S_{14} S_{13}$	S_{11}, S_{10}, S_9	$S_7 S_6 S_5$	(5) ← $S_3 S_2 S_1$
(9)	(9)	(8)	

4 CLA → 5   16 CLA ⟷ 9

64 bit CLA → 13

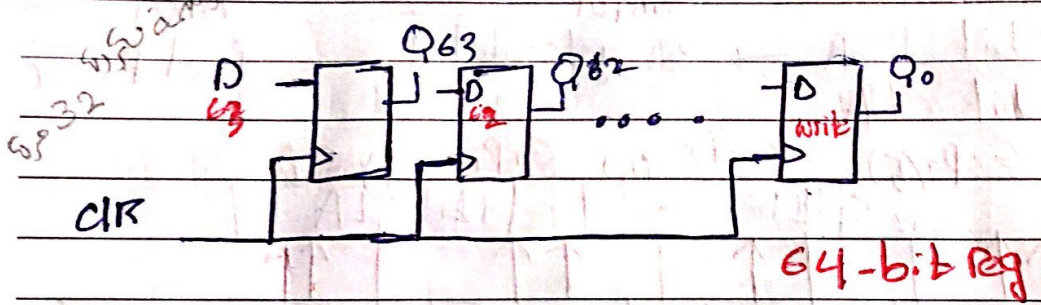
Scanned with CamScanner

2 April 2020
Chapter 4

Fetch instruction → To bring the instruction from the instruction memory to CPU

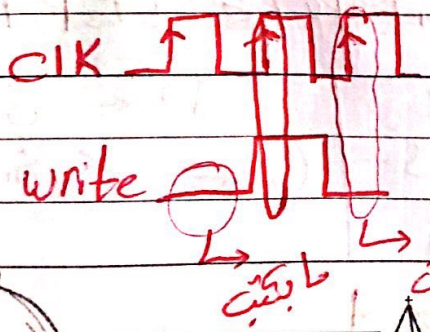
Combinational → output بيعتمد على current input
↳ Add / ALU / multiplexer / control unit

Sequential → بتخزن الـ data
↳ Instruction memory / reg file / data memory

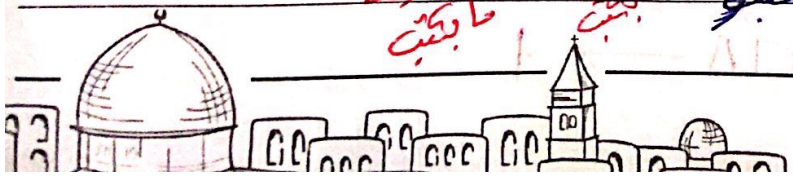


* إذا بدي اقرأ قيمة الـ reg بطلع على قيم الـ Q
$Q_{63} : Q_0$

* اذا بدي اكتب على الـ reg بحط القيمة على الـ D



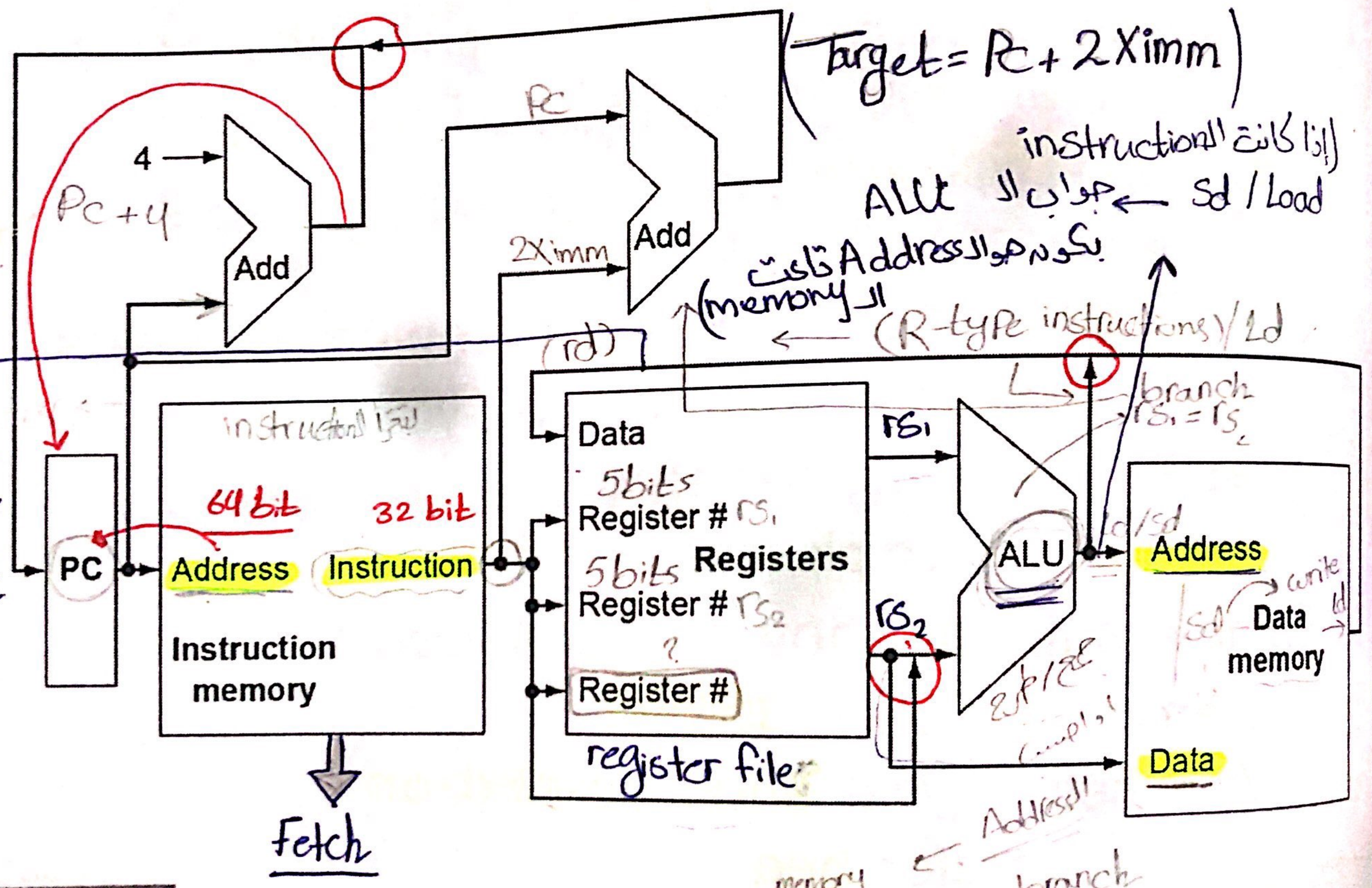
* كل reg عبارة عن 64 D-Flip Flop
add - Sub - Ld ← بكتبوا على الـ reg file
الـ beq / Sd ما بكتبوا



CPU Overview

* Data Path for CPU *

* multiplexers *

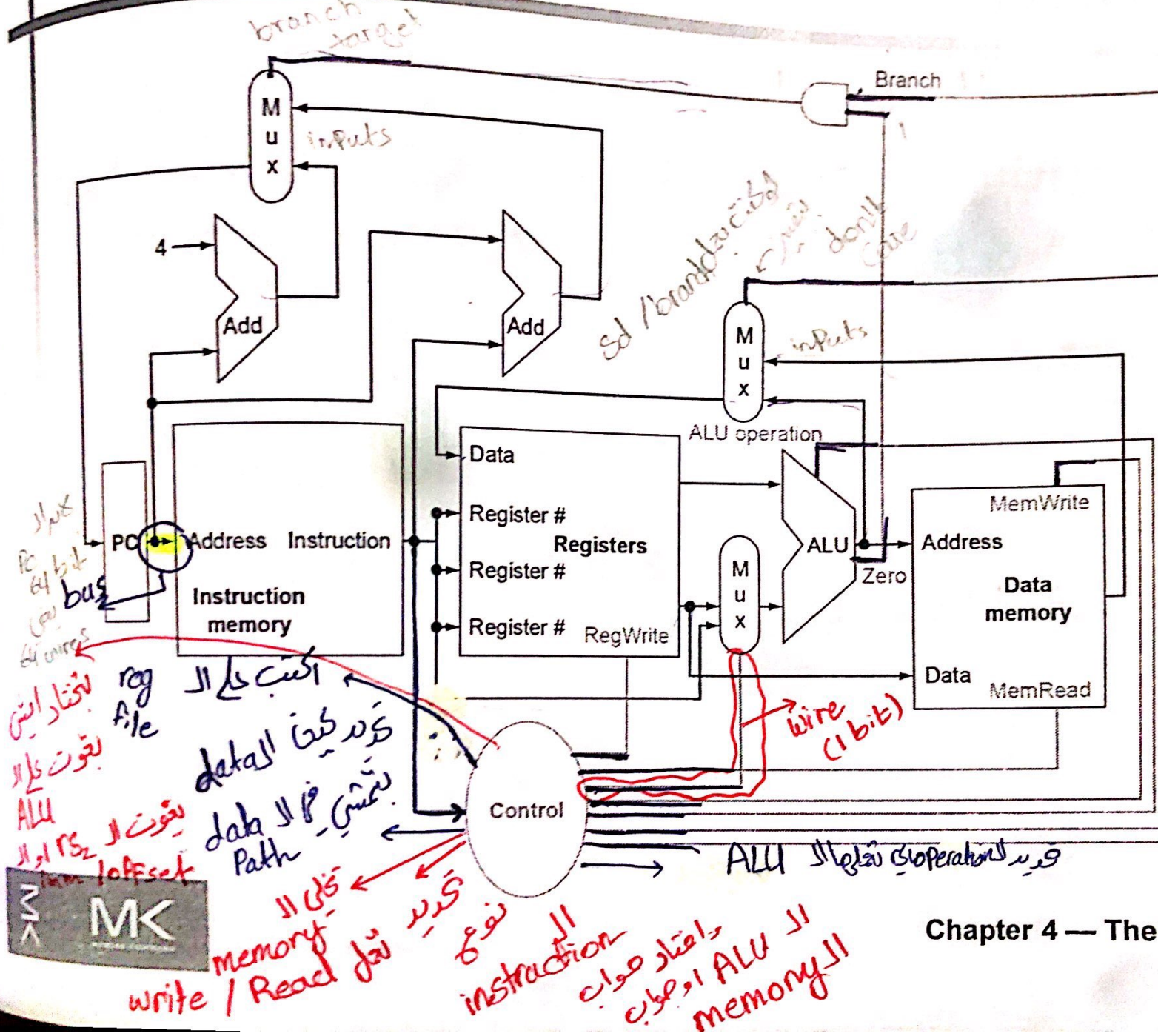


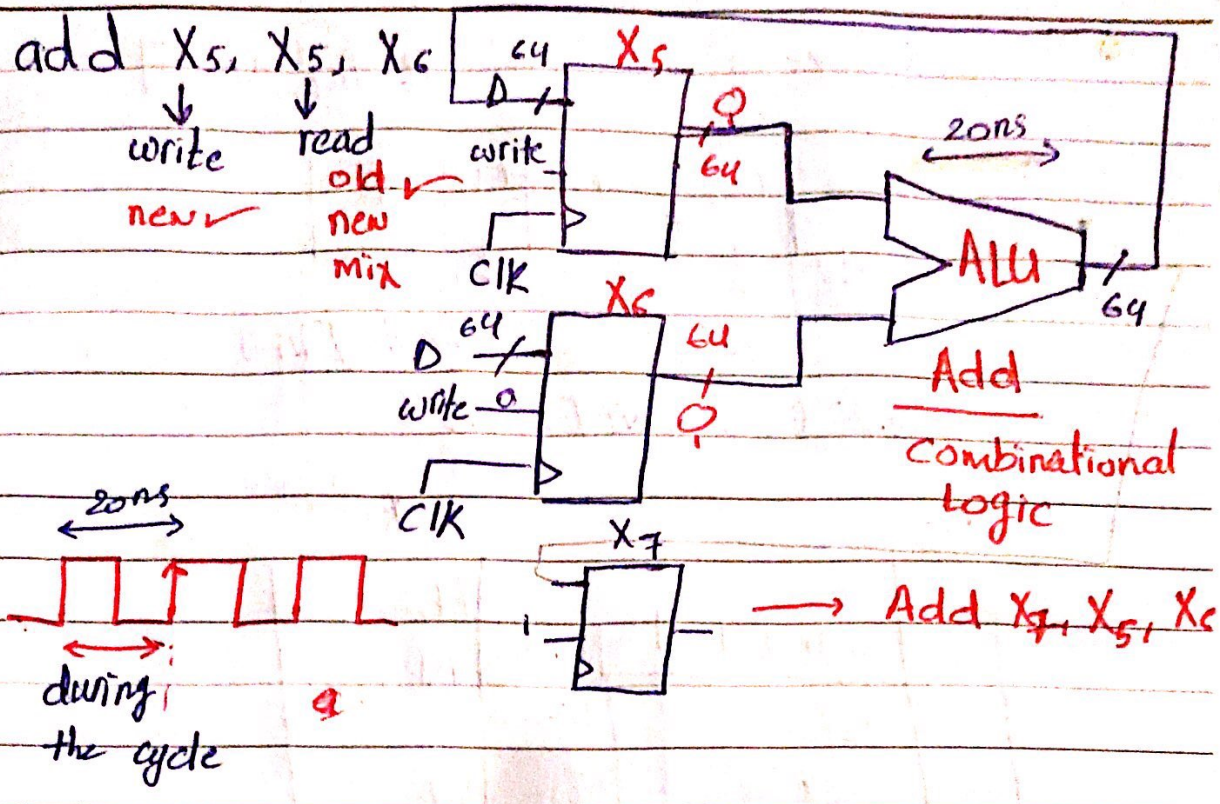
لوكانت العملية من نوع R-type جواب ال ALU حيروح للداتا عشان اكتبه ال rd

إذا كانت ال instruction sd / Load ← جواب ال ALU يكون هو ال Address تاعت ال memory

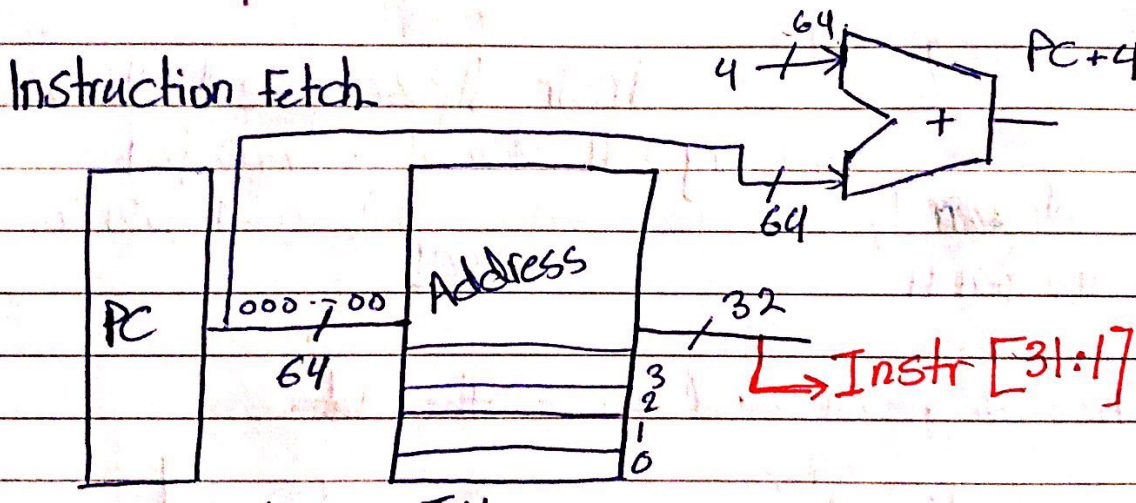
إذا كانت العملية sd ال Address وال Data بعطيها للـ memory عشان تكتب بال memory

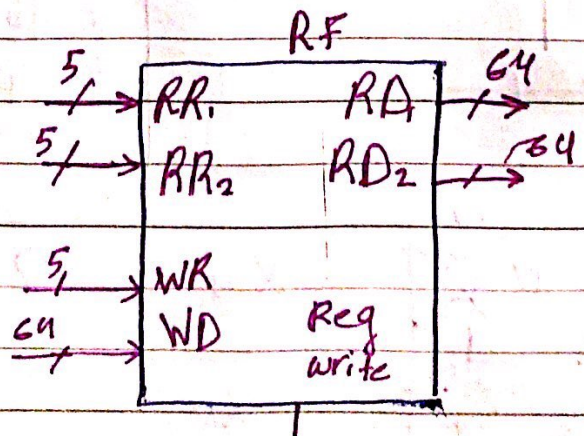
Control





5 April 2020



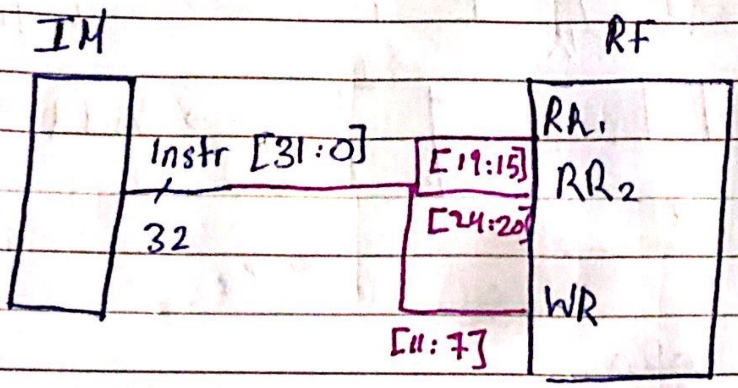


R-Type   Instr [31:0]

funct7	rs2	rs1	funct3	rd	op

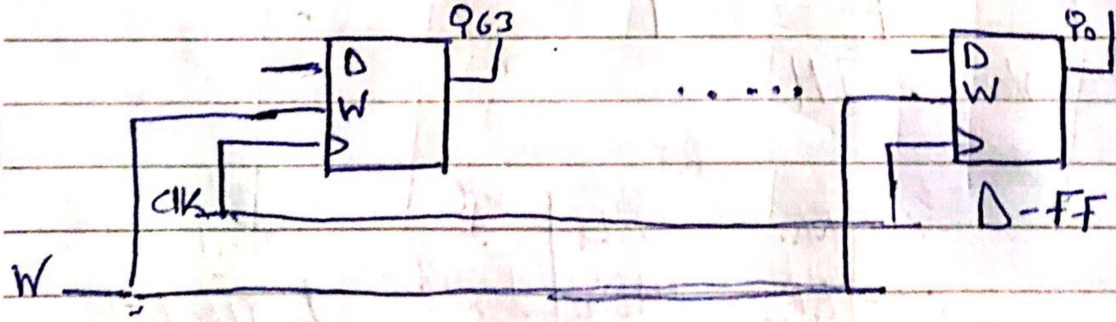
32 bit   Instr [11:7]

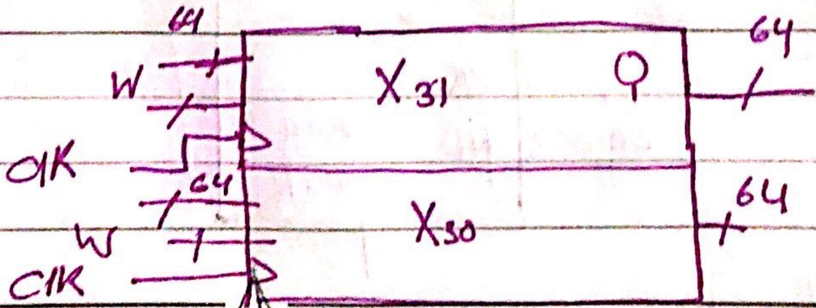
rs2 [24:20] rs1 Instr [19:15]



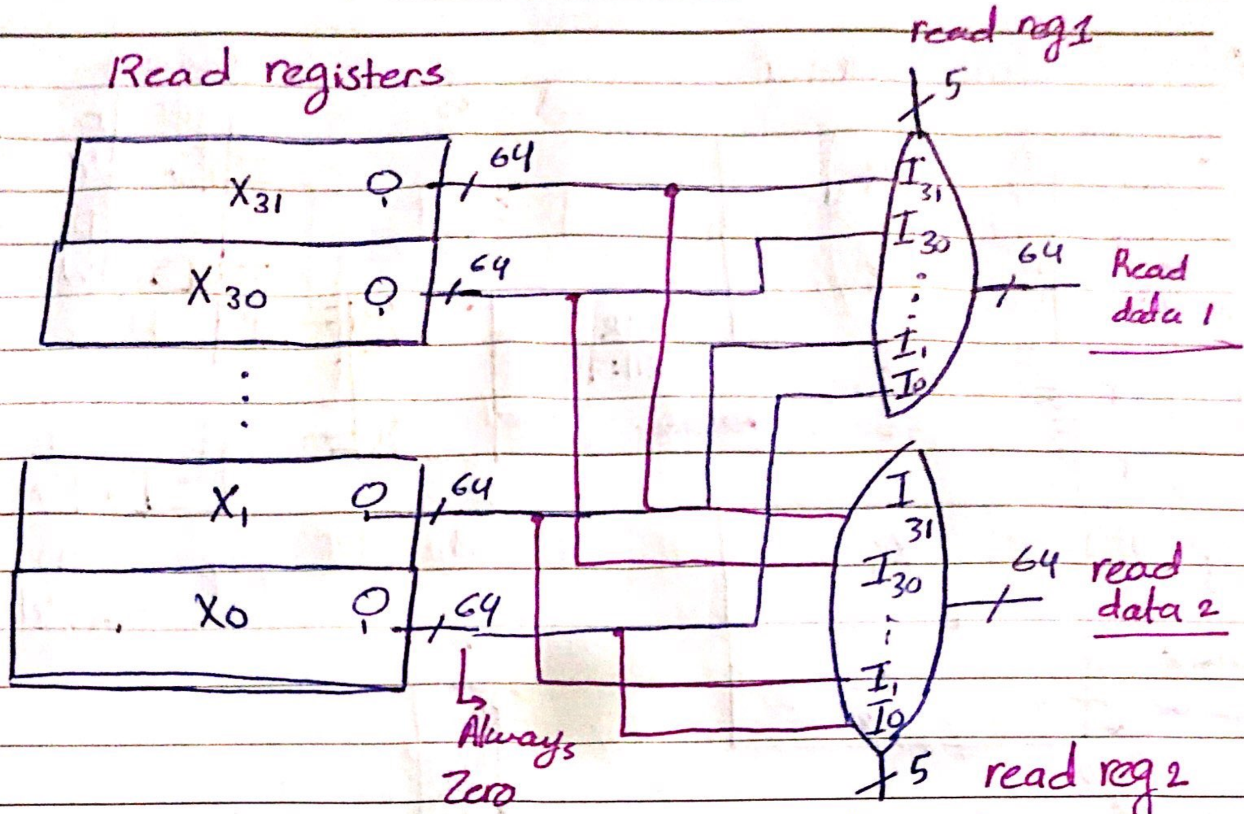
* خلال ال cycle بقرأ ال instr
اصلا هو عشان هيك ال memory
ما بقدر استخدمها ال data memory عشان استخدمها :. خلال ال instruction

How to read and write the data?





Read registers



32 → inputs → 2^5 → 5 → select Line

rs1 rs2

add X_6, X_5, X_7 → X_5

00101 ← ⌋ ⌊→ 00111

X_5 ← I_5 ← rs1 ↵ ⌊→ rs2 → I_7 → X_7

Addi rd, rs1, imm

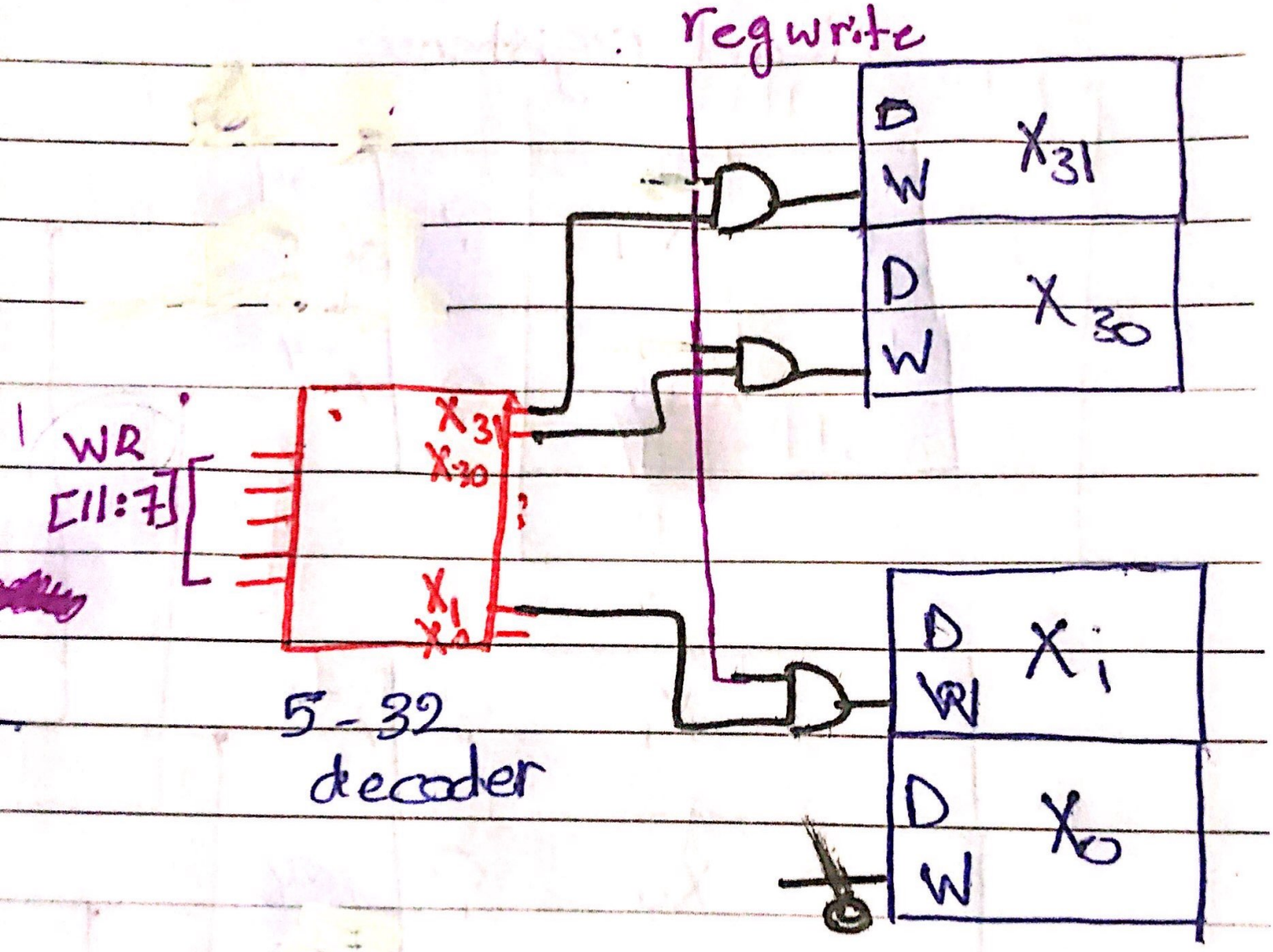
imm	rs	f3	rd	op

⌊→ Instr [19:15]

الحتة كلا ال ، rs 2 ← ما رح اقرأها

كل ال instructions رح يقرؤوا كل ال reg بس عكس استخدم

كلهم او استخدم، احد منهم او ما استخدمهم

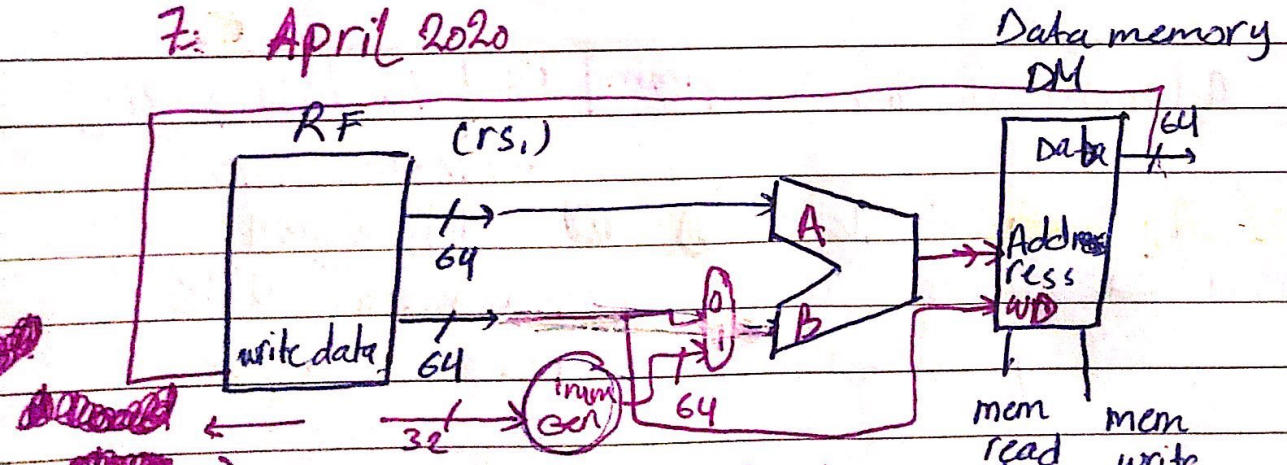


$F_7 = 1$

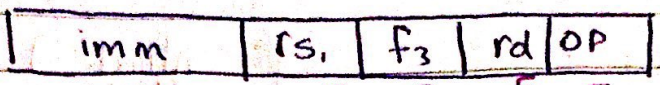
0 111 →

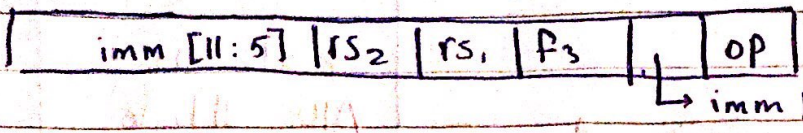
والباقي صفر

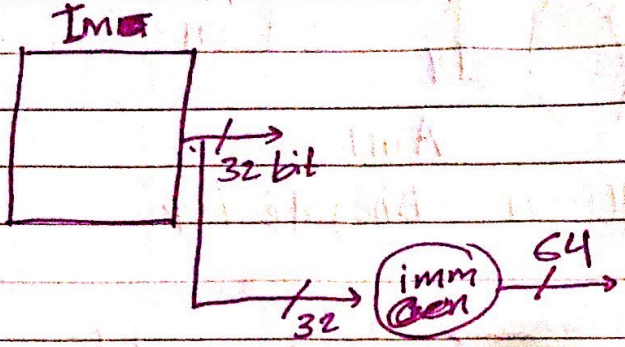
7: April 2020



g. ld rd, offset (rs1)
sd rs2, offset (rs1)
Memory address = (rs1) +







most significant bit

64 ← 52
12

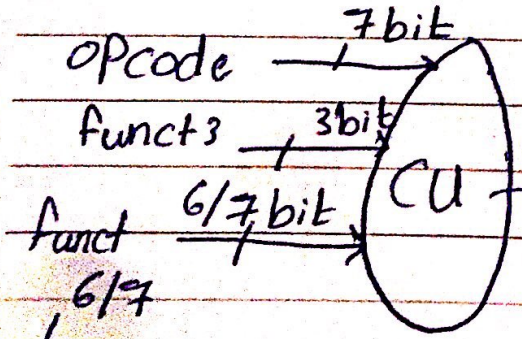
Branch instruction

① Target = PC + 2 × imm

1 - 1 1 1 1 0 — 0 ② 0 ... 0 0 0 1 ... 0
52 52 52

1 ... 11 0000 0000 0000 ← 0
1 ... 11 0000 0000 0000
0001 ---- ---- ---00 2's comp
$-2^{12} = -4096$
1000 0000 0000
$-2^{11} = -2048$

9 April 2020

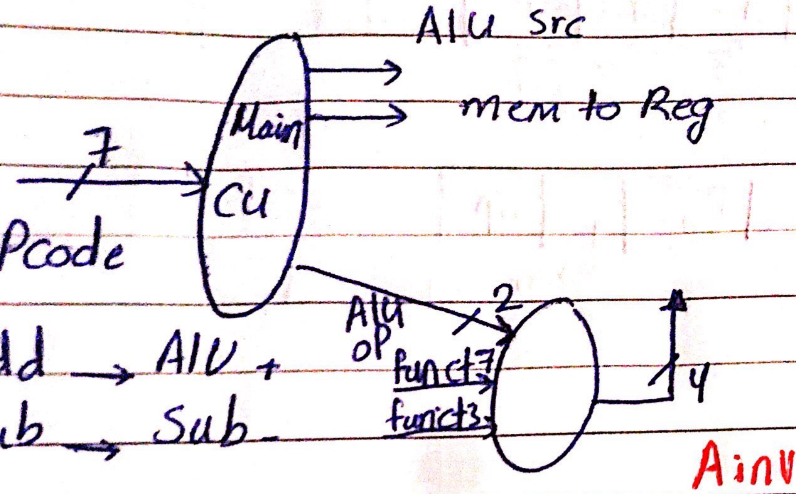


option I

SII: Truth table → zero

2^{17} rows

Option II



add → ALU +
Sub → Sub -

[كل ال Control unit بتعتمد على ال opcode ماعدا ال ALU CU]

Ainv

ALU CU Bnegate / op

ALU-OP

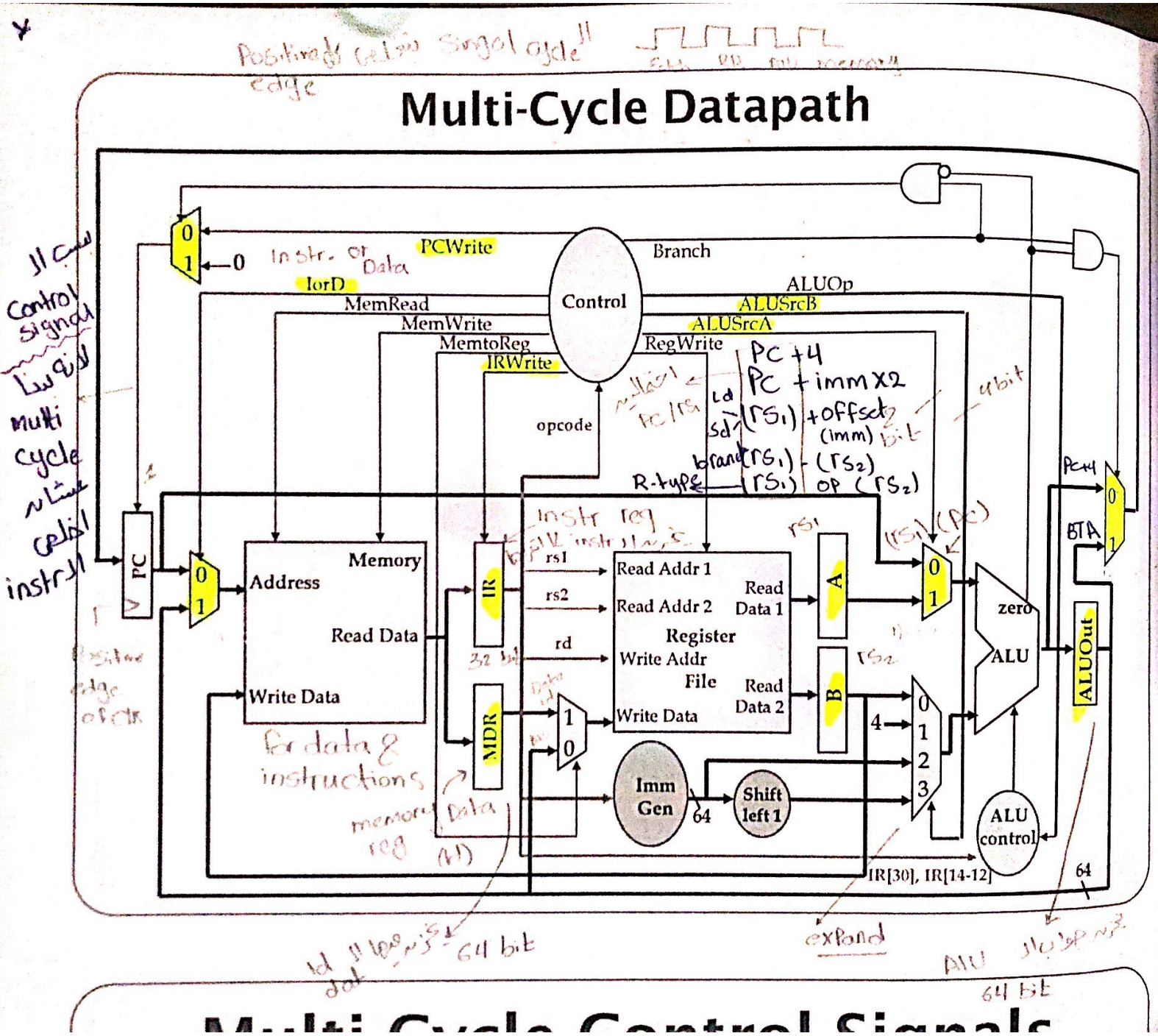
00 (+) → ما بطلع ع ال

01 (-) → branch → funct3 - funct7

10 → [illegible] R-Type → funct7 + funct3 بطلع ع ال

11

Multi-Cycle Datapath



Instruction Execution

- **Cycle 1 – Fetch**
  - **Same for all instructions**
  - Operations
    - Send the PC to fetch instruction from memory and store in IR

Positive edge ← store read
32 bit reg ← **IR ← Mem[PC]**
Mem (Address)

  - Control Signals
    - IorD = 0 **(Select the PC as an address)**
    - MemRead = 1 **(Reading from memory)**
    - IRWrite = 1 **(Update IR)**

IR [31:0]
IR [6:0] → opcode

R-Type IR [11:7]

Instruction Execution

- **Cycle 2 – Decode & register read**
  - Operations
    - Read two registers based on the rs1 and rs2 fields and store them in the A and B registers

Reg File rs1 64 bit
store ← **A ← Reg[IR[19:15]]**
rs2 → we don't need control signal

Positive edge
B ← Reg[IR[24:20]]

    - Use the ALU to compute the branch address

Branch Target Address → **ALUOut ← PC + (sign-extend(Imm) << 1)**

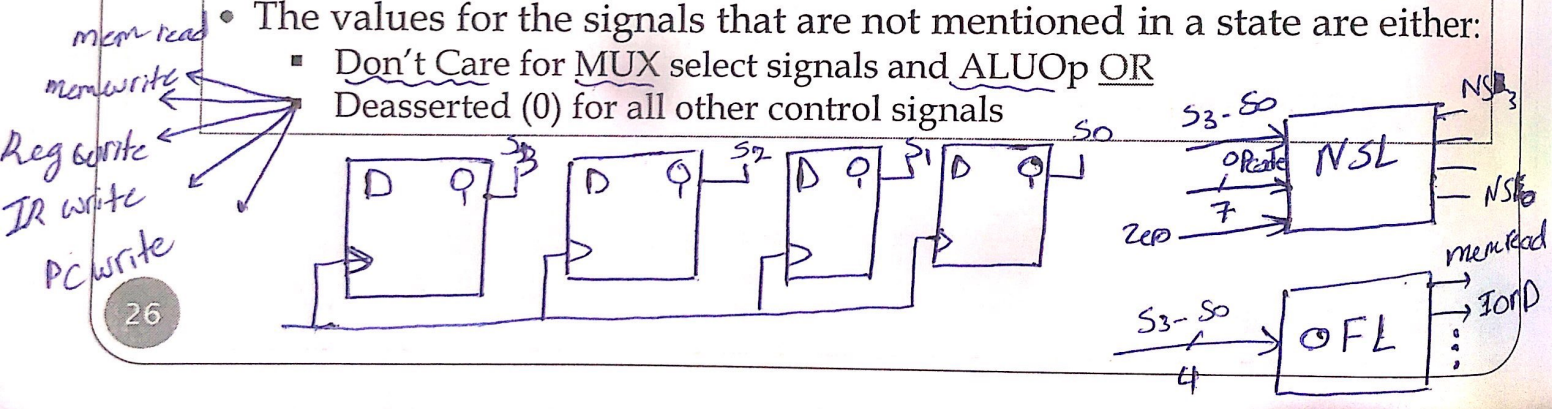
    - Is it always a branch instruction???

ALU Src A = 0
ALU Src B = 11
ALUOp = 00

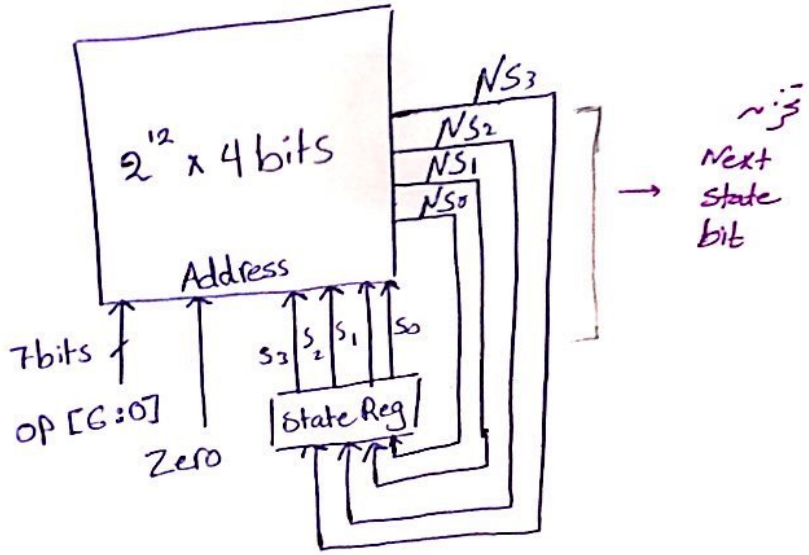
  - Control Signals
    - ALUSrcA = 0 **(Select PC)**
    - ALUSrcB = 11 **(Select the sign-extended offsetx2)**
    - ALUOp = 00 **(Add operation)**

- State memory (SM): 4 bits that represent the current state ($S_3S_2S_1S_0$)
- Next State Logic (NSL): *Combinational logic* responsible for determining the next state as function of current state and input ($NS_3NS_2NS_1NS_0$)
- Output Function Logic (OFL): *Combinational logic* responsible for determining the control signals as function of current state

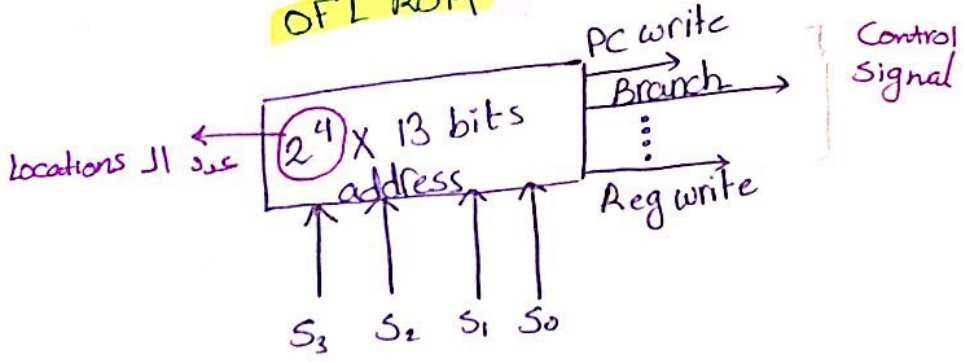
- The values for the signals that are not mentioned in a state are either:
  - Don't Care for MUX select signals and ALUOp OR
  - Deasserted (0) for all other control signals



NSL ROM

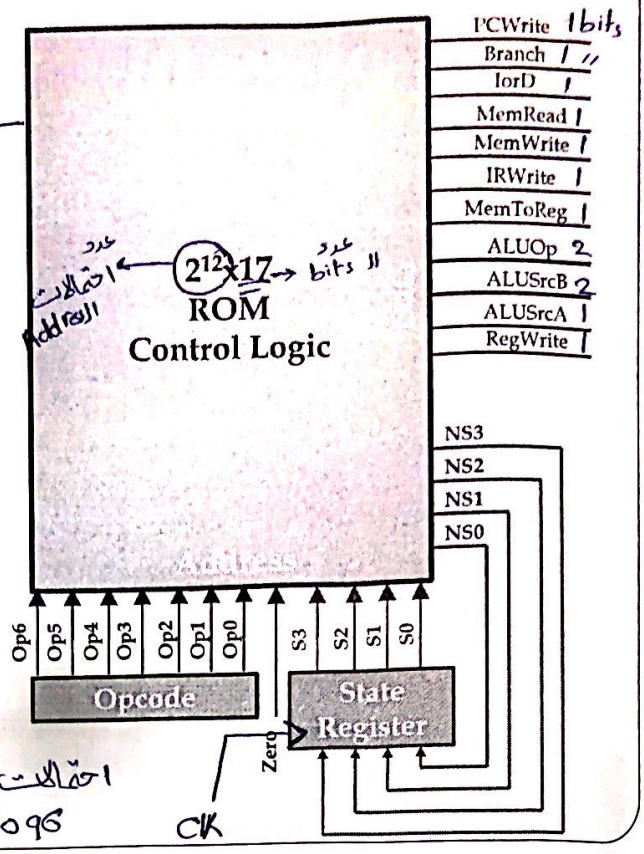


OFL ROM



Multi-Cycle Control

- **(2) ROM Implementation**
  - **FSM design**
    - 12 inputs
    - 17 outputs
    - TT size = 2^{12}x17
  - **ROM**
    - Can be used to implement the truth table above
    - ROM Size = 69632 bits
    - Each location stores the control signals values and the next state
    - Each location is addressable by the opcode and current state value
    - For which ROM addresses will the RegWrite be 1?
      - xxxxxxxx0111
      - xxxxxxxx0100
      - Total of 512 addresses
    - For our design, Only 15 locations are needed → *Huge waste of space*



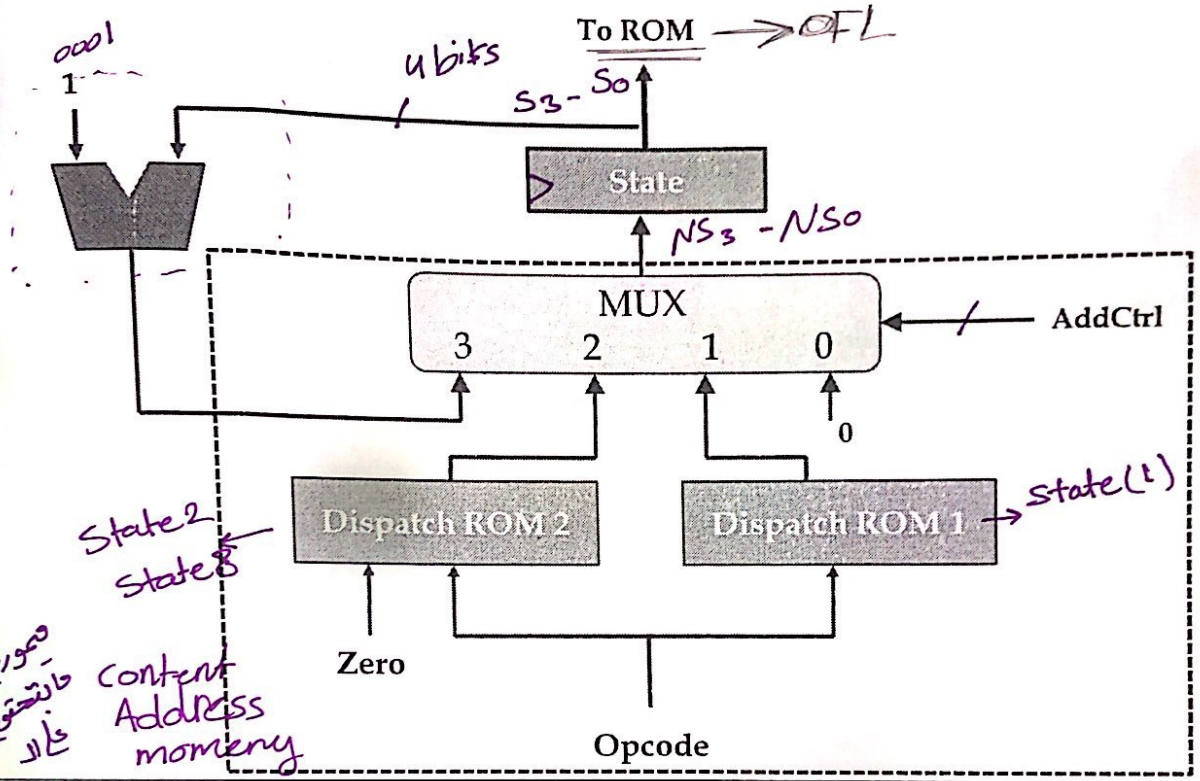
Multi-Cycle Control

- ROM implementation is vulnerable to bugs and expensive especially for complex CPU
- Size increase as the number and complexity of instructions (states) increases
- If we use two ROMs, one for NSL and one for OFL:
  - *NSL ROM* is addressable by the opcode, zero flag and current state value:
    - NSL ROM Size = 2^{12} x 4
  - *OFL ROM* is addressable by the current state value:
    - OFL ROM Size = 2^4 x 13
  - Total Size = 16592 bits (much lower than a single ROM)
  - NSL consumes approximately 99% of the ROM area
- **Solution: Implement the Next State Function with a Sequencer**

Multi-Cycle Control

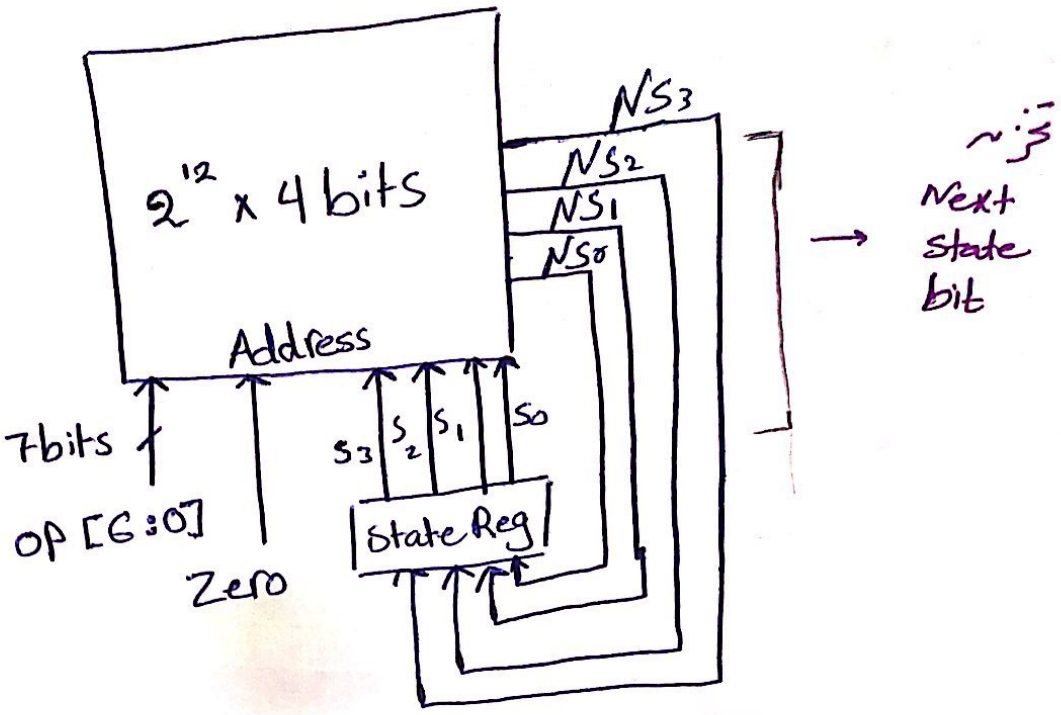
- **Implementing the Next State Function with a Sequencer**

Inside the address select logic

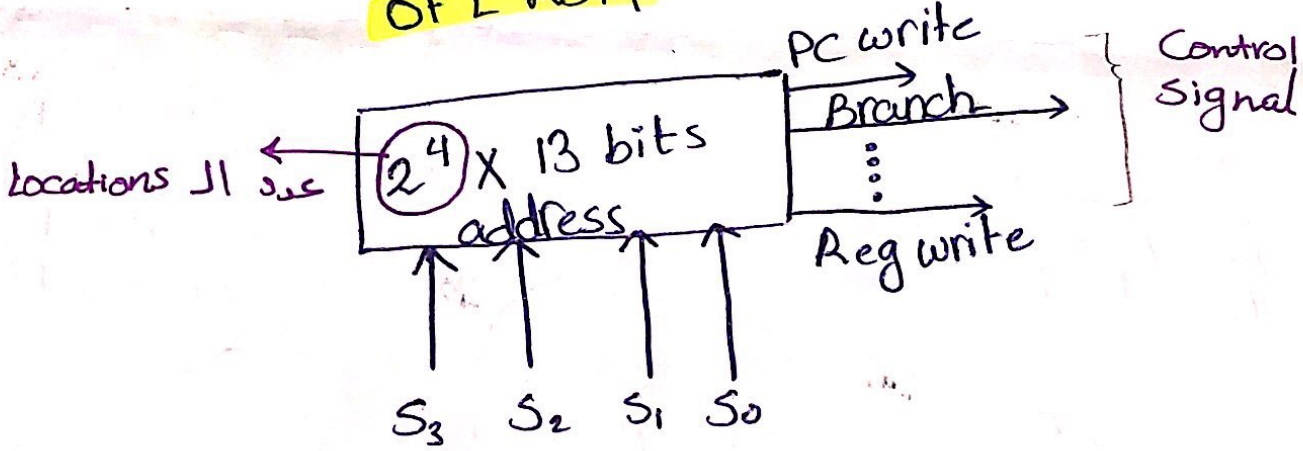


Multi-Cycle Control

NSL ROM



OFL ROM



RISC-V Pipeline

5 stages Pipline

- Five stages, one step per stage
  1. IF: Instruction fetch from memory (F)
  2. ID: Instruction decode & register read (D)
  3. EX: Execute operation or calculate address — Execute ALU
  4. MEM: Access memory operand — Ld/Sd (Address read or write memory)
  5. WB: Write result back to register (rd)

IM, DM → 2 separate memory — Single cycle ... Ls ...

- Stages can operate concurrently as long as (separate resources) are available for each stage
  - (Pipeline design is based on Single-Cycle CPU design)

Pipeline

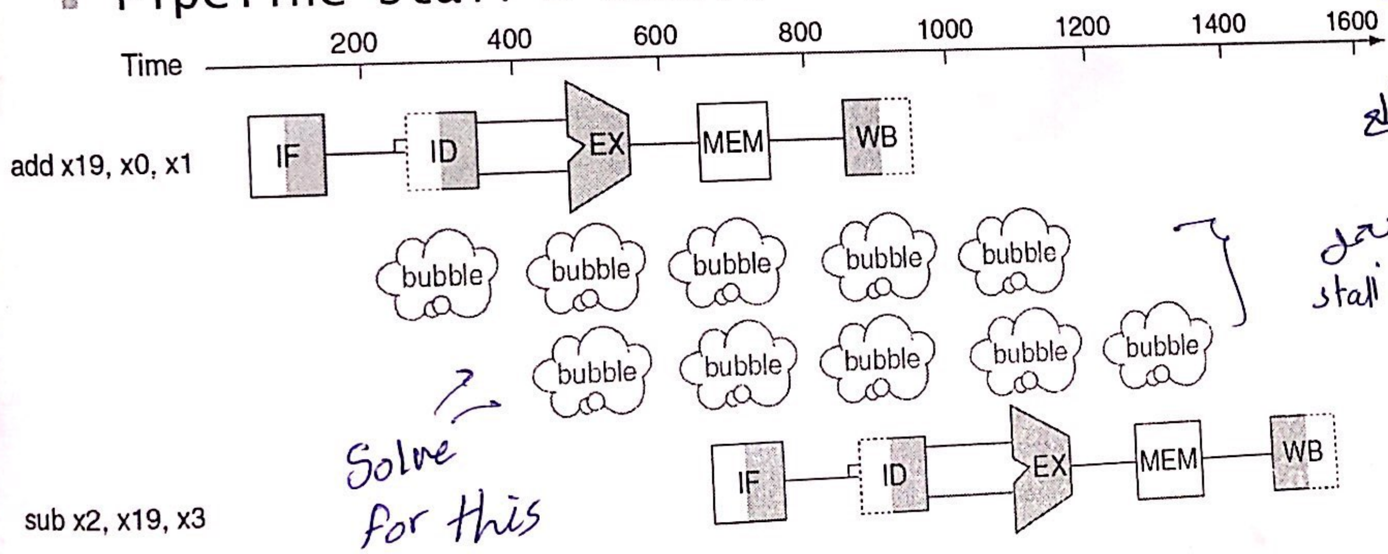
Same design Single cycle

Same Clock Period (multi cycle)

Pipeline Performance

Data Hazards

- An instruction depends on completion of data access by a previous instruction
  - add x19, x0, x1
    sub x2, x19, x3
- Pipeline must be stalled (i.e. stopped)
  - Pipeline stall ≡ bubble



Load-Use Hazard Detection

- Check when using instruction is decoded in ID stage
- ALU operand register numbers in ID stage are given by
  - IF/ID.RegisterRs1, IF/ID.RegisterRs2
- Load-use hazard when
  - ID/EX.MemRead and (ID/EX.RegisterRd ≠ 0) and ((ID/EX.RegisterRd = IF/ID.RegisterRs1) or (ID/EX.RegisterRd = IF/ID.RegisterRs2))
- If detected, stall and insert bubble