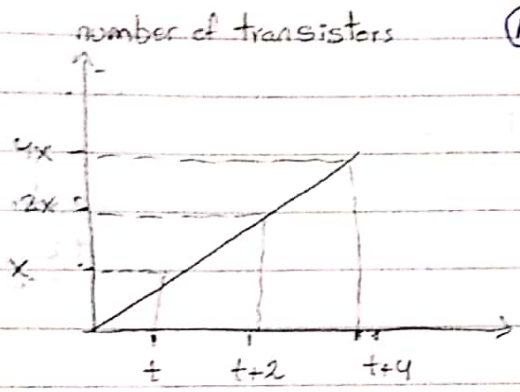
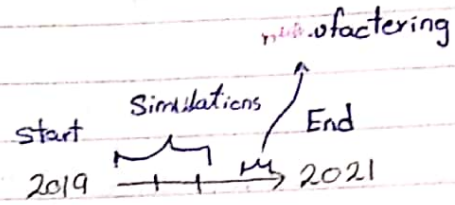
No: ..... Date: 24/9/2019

Eight Great Ideas :

* number of transistors perchip.



(A) Design for Moore's law



every 18 month - 24 month

Classes of computers : (A) Supercomputer :

* Note : * top 500.org → Supercomputer أقوى ١٠٠ موقع للإطلاع :

← T Flop / sec

Tera ↵ ↳ Floating point ↳ operation.

(B) Embedded computer :

Power / Performance / cost

قل ↓ زاد ↑ ↓

* Instruction set Architecture " ISA " (Hardware/software) أداة الربط بين

MIPS Intel x86 ARM RISC - V ("أضعف")

* understanding performance

① Algorithm "Sorting" : number of operations to be executed

② Programming language, compiler, ISA : number of machine instructions to be executed

③ Processor and memory

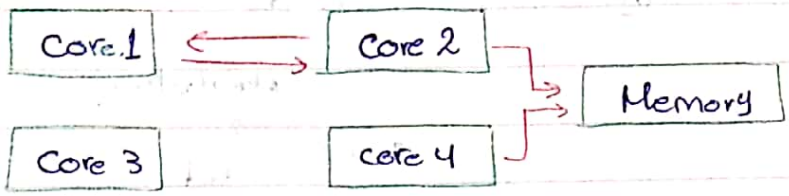
④ I/O

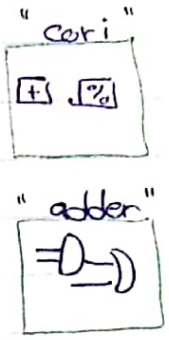
→ "org" من "design" كل

YASSIN

No: Date: 26/9/2019

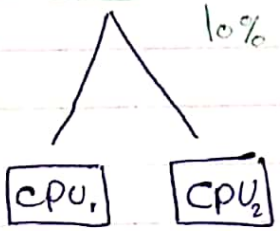
Ⓑ Use a subtraction to simplify design





Ⓒ Make the common case Fast

Ex: CPU: 90% addition — add (1ns)
10% division — divide (10ns)



add 0.2 nsec add 1 nsec
divide 10 nsec divid 5 nsec ↷ "nano second"

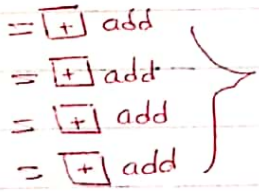
* الأفضل نسرع الـ "common case" عشان يطلع الـ Time أقل.

100 operation ↗ 90 add
↘ 10 divide

Time CPU_0 : 90 x 1 + 10 x 10 = 190 ns
Time CPU_2 : 90 x 1 + 10 x 5 = 140 ns
Time CPU_1 : 90 x 0.2 + 10 x 10 = 118 ns

Ⓓ Performance via parallelism (يشتغلوا عالتوازي)

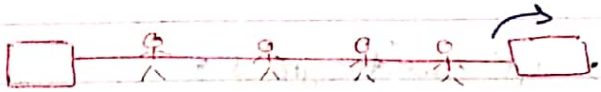
Ex:



4 Programms (يشتغلوا عالتوازي)
* لما نحطلهم مع بعض عالتوازي بقل الزمن.

Ⓔ Performance via Pipelining

Ex:



تقسيم خط التنفيذ أكثر من قسم لتسهيل وتسريع مراحل العمل وهو جزء من الـ Parallelism

* كل ما تخلص مرحلة وتروح للي بعدها بنفس ترجع تستخدم المرحلة اللي قبل.

No: ---------- Date: ----------

(F) Performance via prediction

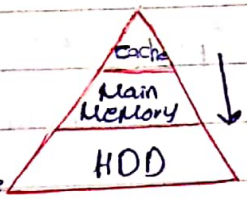
Ex:

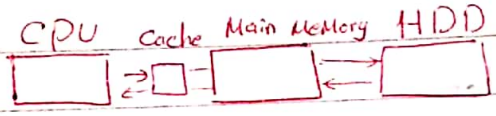
```
if (condition)
{

}
else
{

}
```

لهذه الطريقة نستخدم الشيء قبل معرفة نتيجة الـ condition إذا كان False / true . فمثلاً نتنبأ أن النتيجة True وتقوم بتنفيذ جملة الـ "IF" ونسبة نجاح هذه الفكرة غالباً تتعدى 50% ولكن إذا كانت أقل من 50% فهي غير مجدية . وإذا 50% بدون فائدة .

(G) Hierarchy of memories (تخزين الـ data في أكثر من مكان)

Ex:





→ بهاي الحالة صارت أسرع لأنه ياخذ الـ data من الـ Cache والـ Main

access time ↓ الزمن اللي يحتاجه لوصول الـ data من مكان لآخر

cost / speed / size ↓ decrease ↓ decrease ↑ increase

↳ increase

(H) Dependability via redundancy. (تخلي أكثر من نسخة للـ component للاحتياط) (بنستفيد منها في حال خربت وحدة مش كشان نحسن الـ cpu)

=[+]
=[+]

1.3

system software → compiler: translator
→ operating system (os)

29/9/2019

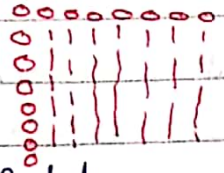
Componant of a computer:

① Input
② output → User-Interface: Display, Storage devises, Network adapters
③ Memory
④ Datapath → ⑤ Controlunit → Processor "cpu"

YASSIN

No:---------- Date:----------

* bit map

Ex :  "8-bit"

Size of bit map :

① Screen Size .

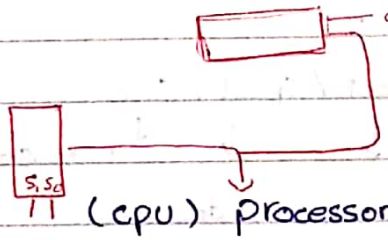
② Resolution .

* buffer : ⟶

* Datapath : Example of datapath component : * MUXes * Decoders

* Adder/ subtractors

* control unit :

(control unit) 

(cpu) processor هو بعض هم اللي بكونوي ال

* Apple A5 :

A5

* A safe place For Data

The difference between : cache : static RAM / 6-invertes (أسرع)

Main Memory : Dynamic RAM / one capacitor

(أبطأ)

* ال Memory ما بتوسع كلشي بتخزن المعلومات أو ال data المستخدمة حالياً في العمل

و إذا طفيت ال Computer بتفل الداتا مخزنة في ال Magnetic disk

No:---------- Date:----------

Storage Devices

	DRAM	Magnetic disk	Flash
Price/GByte	Expensive 5$	cheap (0.05% - 0.1%)$	Moderate (0.75% - 1%) $
Speed	Fast (50-70) ns	slow (5-20) ms	Moderate (5-50) µs
Volitility	Volatile	Non-Volatile	Non-Volatile
Wearout معناها إنه الاشي تخرب أو يبطل مستخدم	N/A	N/A	1,00,000 - 1,000,000 writes

1.5 & Defining performance :

* time هو المرتبط بالـ performance *

Response (Execution) (Elapsed) (wall clock) time : الزمن اللي بتحتاجه من البداية للنهاية لأداء الـ Task وتنفيذها.

Throughput : مقدار الوظائف اللي سويتها خلال وقت معين

Ex: Data center
- task: 10 ns
- throughput: 60 tasks / 60 ns

مين أحسن ؟ الثانية أحسن من الأولى لأنها تؤدي وظيفة واحدة بـ 10 ns بينما الثانية تؤدي 60 tasks خلال 60 ns

Ex: speed of CPU_2 : 2x speed of CPU_1

CPU_1 → CPU_2
- Response time → decrease by half
- throughput → doubles

العلاقة بينهم عكسية دائماً

CPU_1 → CPU_1 CPU_1
- Response time → stays the same
- throughput → doubles

* أمّا في حال قسمت الـ task بالنص فـ Response time بقل بالنص

No: ---------- Date: ----------

* Performance = $\frac{1}{\text{Execution time}}$ ، X is n times Faster than Y

↳ العلاقة بينهم عكسية دائماً

$$\frac{\text{Performance X}}{\text{Performance Y}} = n = \frac{ET_Y}{ET_X}$$

Ex : CPU A = 10 Sec → better than B

CPU B = 15 Sec

$$\frac{ET_B}{ET_A} = \frac{15}{10} = 1.5$$

↳ مقدار الزيادة عن 1

A is 1.5 times Faster than B => A is 0.5 × 100 % = 50% faster than B

---

1 / 10 / 2019

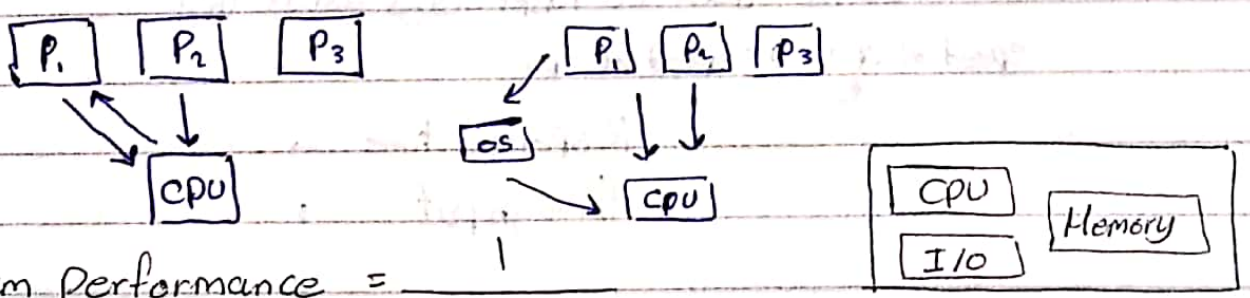
Ex : CPU A = $\frac{1}{15}$ Sec

CPU B = $\frac{1}{5}$ Sec

$$\frac{\frac{1}{5}}{\frac{1}{15}} = \frac{15}{5} = 3$$

A is 3 time faster than B => A is 2 % faster than B

* idle time : الوقت اللي منتظر بدون ما يعمل اشي



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

cpu time :

- User cpu time ← الوظائف اللي ماينتظل فيها operating system الـ
- System cpu time ← الوظائف اللي بتدخل فيها operating system الـ

$$\text{cpu Performance} = \frac{1}{\text{User cpu time}}$$

Ex : P_1 → cpu time = 10 ns

User cpu time	system cpu time
7 ns	3 ns

لـ مثلاً كالوظيفة التالية :

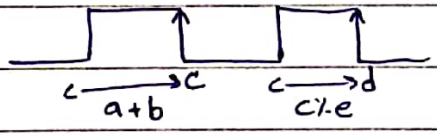
```
C = a+b ;
D = a%e ;
!
```

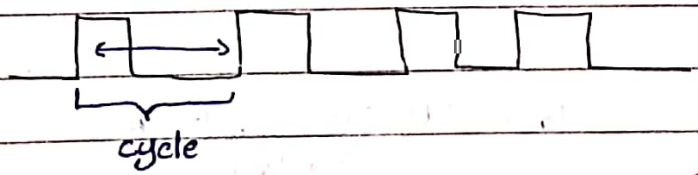
cpu : Sequential system → يعني بتخزن البيانات اللي قبلها وبستخدمهم

↳ cpu : synchronous sequential system

EX : C = a+b ; ⇒

D = c%e ;





Clock cycle time = Clock Period = 250 ps = 0.25 ns (Ex ← 10^{-12})

Clock frequency = number of cycles per second (HZ) (التكرار) =

Ex : $\frac{1 \text{ sec}}{250 \times 10^{-12} \text{ sec}} = \frac{10^{12}}{250} = \frac{1000 \times 10^9}{250} = 4 \times 10^9 \text{ HZ} = 4 \text{ GHZ}$

↳ clock rate تسمى أيضاً

No: ---------- Date: ----------

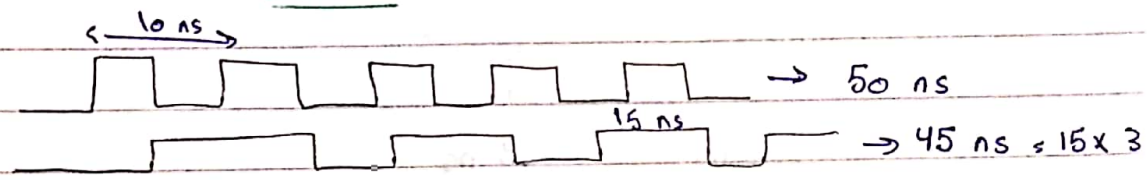
User CPU time = CPU clock cycles X Clock cycles time

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

* user cpu performance = $\frac{1}{\text{user cpu time}}$

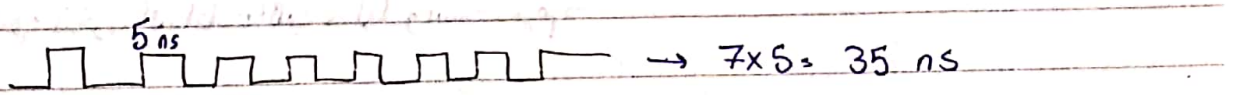
↳ رح أصيرها أكتب كلمة user لأنو معروف إنه اللي بستخدموا « user CPU »

Ex:



* هون قلت الـ " cycles " بس زادت clock cycle ولكن قل cpu time ↑.

* هون زدت الـ " cycles " بس قلت clock cycle والـ cpu time قلت زيادة وهو أفضل حل ↓



Ex: $F_A = 26$ Hz

$CPU\ time_A = 10$ seconds

CPU tim B = 6 sec

cpu clock cycles B = 1.2 X cpu clock cycles A

$F_B = ??$

Sol: $CPU\ time = \frac{\text{CPU clock cycles}}{\text{clock rate}} \Rightarrow 10 = \frac{\text{CPU clock cycles A}}{2 \times 10^9}$

CPU clock cycles A = 20×10^9 cycle

∴ cpu clock cycles B = $1.2 \times 20 \times 10^9 = 24 \times 10^9$ cycles

∴ $6 = \frac{24 \times 10^9}{\text{Clock rate B}} \Rightarrow$ clock rate B = $\frac{24 \times 10^9}{6} = 4 \times 10^9$ Hz = 4 GHz

↳ جيجاهيرتز

→

YASSIN

* cycle $time_A = \frac{1}{F} = 0.5$ sec
* cycle time $B = \frac{1}{F} = \frac{1}{4} = 0.25$ sec → . هو الأسرع والأحسن

* $\frac{\text{Performance B}}{\text{Performance A}} = \frac{\frac{1}{6}}{\frac{1}{10}} = 1.67$

B is 1.67 times faster than A
B is 67% times faster than A

* CPU clock cycles = Instruction Count X Clock per Instruction
= IC X CPI
↳ عدد الـ Instruction ↳ كم بدي الـ cycles للـ instruction الواحدة

* CPU time = clock cycles X clock cycle time
* CPU time = IC X CPI X clock cycle time = $\frac{IC \times CPI}{\text{Clock rate}}$

Factors of IC

- Program
- ISA ↳ Instruction set
- Compiler

3/10/2019

CPU time = CPU clock cycles X clock cycle time
CPU clock cycles = IC X CPI
CPU time = IC X CPI X clock cycle time = $\frac{IC \times CPI}{\text{clock rate}}$

IC : Program, ISA, compiler
أساس

C = a*b + d;
MAD (multiplication and add) : 1 instraction
multiply ⎫ 2 instruction (بحتاج 2 مش 1 لأنهم
add ⎭ ورا بعض)

YASSIN

Ex: HLL → c = a * 3 ;

↓ compiler

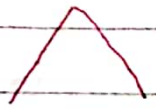
Assembly

Compiler 1: add t,a,a ; / add c,t,a ;

Compiler 2: multiply c,a,3

Ex 2: Sorting

3,4,2,0,7,5



HLL HLL

10 lines 30 lines

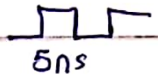
IC_1 ≠ IC_2

compiler: يحول من HLL الى Assembly ← high livle language

*CPI: cycle per Instruction.

Single cycle cpu → "instruction" تنفيذها حسب أبطأ

Ex: compiler 1: 5 ns → 

compiler 2: 20 ns → 

→ multi cycle cpu

* add → 1 cycle

* multipl → 4 cycles

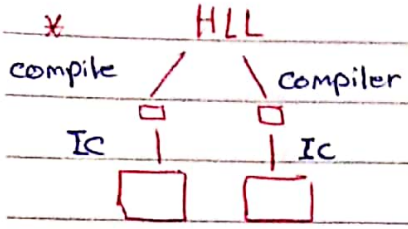
* $CPU = \frac{1+4}{2} \rightarrow CPU_{avg} = 2.5$

* 30 add / 70 multiply → CPU_{avg} بال حسب تسارع

No: ---------- Date: ----------

Examples :

* CPU time = CPU clock cycles X clock cycle time
* = IC X CPI avg X clock cycle time
Instruction count



* Same ISA ← أيضا → Same compiler (Page 33 راجع)

EX Page-33 : CPU time A = $IC_A \times 2 \times 250 \times 10^{-12} = 500\, IC_A \times 10^{-12}$

CPU time B = $IC_B \times 1.2 \times 500 \times 10^{-12} = 600\, IC_A \times 10^{-12}$

$$\frac{\text{CPU time B}}{\text{CPU time A}} = \frac{600\, IC_B \times 10^{-12}}{500\, IC_A \times 10^{-12}} = 1.2$$

= A is 1.2 times faster than B

= A is 20% times faster than B

Same ISA and same compiler زيدي على السؤال ← Page 33

Ex : Instruction | add | Mul | load

CPI | 5 | 10 | 20

* لاحظ معلومات كفاية لحساب ال CPU Avg والمعلومة الإضافية التي يستخدمان للحل

Relative frequency | 50% | 20% | 30%

* Arithmetic Average CPI = $\frac{5+10+20}{3} = \frac{5}{3} + \frac{10}{3} + \frac{20}{3} = 11.67$

* Weighted Average CPI = $5 \times 0.5 + 10 \times 0.2 + 20 \times 0.3$

$= 2.5 + 2 + 6 = 10.5$ ← وهو الحل الأفضل

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

* ممكن نجيب الـ relative frequency الـ تطلع في الـ IC

Ex: IC 5 2 3

↳ بدي أطلع الـ relative frequency ⇒ $\frac{IC_i}{IC_{total}}$ / $\frac{5}{10}$, $\frac{2}{10}$, $\frac{3}{10}$

* $CPI_{avg} = \frac{\text{CPU clock cycles}}{IC_{total}}$ ← weighted Avg. لازم تطلع بنفس

* $\text{CPU clock cycle} = IC_{total} \times CPI_{avg} = IC_{total} \times \sum_{i=1}^{n} CPI_i \times \frac{IC_i}{IC_{total}}$

* $\text{cpu clock cycle} = \sum_{i=1}^{n} CPI_i \times IC_i$

Ex:

Instruction	add	multi	load
CPI	5	10	20
IC	5	2	3

evalute cpu clock cycles ?

$= IC_{total} \times CPI_{avg} = 10 \times 10.5 = 105$

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

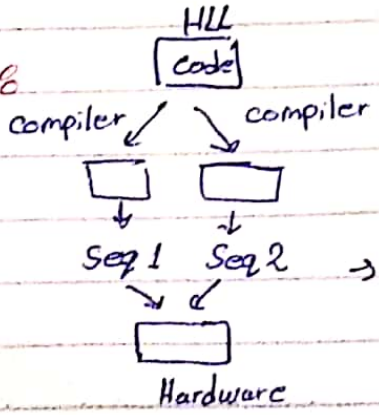
$= 5 \times \frac{5}{10} + 10 \times \frac{2}{10} + 20 \times \frac{3}{10} = 10.5$

or

cpu clock cycle = 5x5 + 10x2 + 20x3 = 25 + 20 + 60 = 105 (هيك أسهل)

* إذا أعطاني بدل الـ IC الـ relativ frequency ← بقدر أحسب الـ "CPIavg" بس

Ex page 35 :



هون بقارن الـ compilers لأنه الـ Hardware نفسه بالنهاية ←

No: .... Date: ....

CPU clock cycles ← موجودين

* CPU_{time} Seq 1 = $(IC_{total} \times CPI_{avg})$ × clock cycles time = $\sum(CPI_i \times IC_i)$ × clock cycle time (ما الك معلومات بس ما دام نفس الـ Hardware بلشوا بعض)

CPU_{time} Seq 1 = 1×2 + 2×1 + 2×3 = 10

* CPU_{time} Seq 2 = 1×4 + 2×1 + 3×1 = 9 → اللي أقل هو الأفضل

* لازم يعطيك معلومة بإنهم نفس الـ Hardware

CPI_{avg} Seq 1 = $\frac{10}{5}$ = 2

CPI_{avg} Seq 2 = $\frac{9}{6}$ = 1.5

Scanned by CamScanner

CPU time = Cpu clock cycles x clock cycle time

CPU time = IC x CPI avg x clock cycle time

cpu time = $\left(\sum_{i=1}^{n} IC_i \times CPI_i\right)$ x clock cycle time

* $IC \times CPI_{avg} = \sum_{i=1}^{n} IC_i \times CPI_i \Rightarrow$ $CPI_{avg} = \sum_{i=1}^{n} CPI_i \times \frac{IC_i}{IC}$

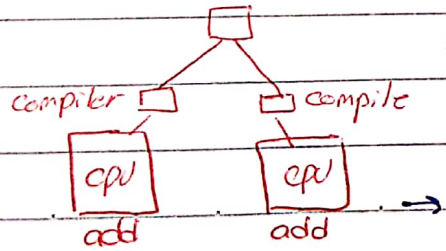
Relativ frequency

* $CPI_{avg} = \frac{\text{CPU clock cycles}}{IC}$

* Note & Same ISA + Same compiler ⇒ Same IC_i and Same IC_{total}

* Note & Same hardware (cpu) => - Same clock cycle time (clock Rate)

- Same CPI_i



CPI_{total} لـ بس عشان نفس

CPI_{total} / CPI_{avg} بس عشان ضروري نفس CPI_i نفس الـ

YASSIN

No: ---------- Date: ----------

Solution example Page 36 :

$$\frac{\text{Performance 1}}{\text{performance 2}} = 2 = \frac{\text{CPU time 2}}{\text{CPU time 1}} = \frac{\text{CPU clock cycles 2} \times \text{clock cycle time}}{\text{CPU clock cycles 1} \times \text{clock cycle time}}$$

$$= \frac{\sum_{i=1}^{3} IC_i \times CPI_i}{\sum_{i=1}^{3} IC_i \times CPI_i} = \frac{2\times1 + ?\times2 + 2\times3}{1\times1 + 2\times2 + 4\times3} = 2$$

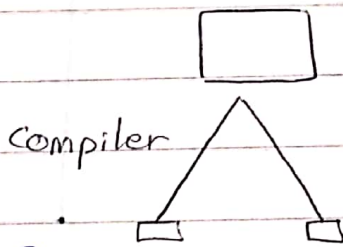
* Summary :

$$CPU\ time = IC \times CPI_{avg} \times \text{clock cycle time}$$

$$= \frac{\text{Instruction}}{\text{program}} \times \frac{\text{clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{cycles}} = \text{Seconds / program}$$

* $$CPI_{avg} = \frac{\text{CPU clock cycles}}{IC_{total}}$$

Chapter 2 :



RISC $IC_{RISC} > IC_{CISC}$ CISC

RISC	CISC
- Fixed length	- variable length
- Simple functionality	- complex functionality
- Fast execution	- Slow execution

* Arithmetic operation.

* ما بنفع إلا 3 opperans بس. ←

Sources ←  → destination (المكان الي بخزن عليه)

Ex : add a,b,c ⇒ a = b+c

a = b+c+d+e ;

↓ compiler

هون المكان اللي بخزن عليه →

add a, b, c ⇒ a = b+c ;
add a, a, d ⇒ a = a+d ;
add a, a, e ⇒ a = a+e ;

* نفس الـ operant بنفع يكون Source, destination.

* إذا بدي الـ better performance لازم يكون نظامية أعلى عشان يكون Simple.

Ex : add t0, g, h
add t1, i, j → هنا بياخذ more temporery Sources
sub f, t0, t1 ; ⇒ f = t0 − t1

الرقم اللي بدك تطرح منه ← الرقم اللي حتطرحه

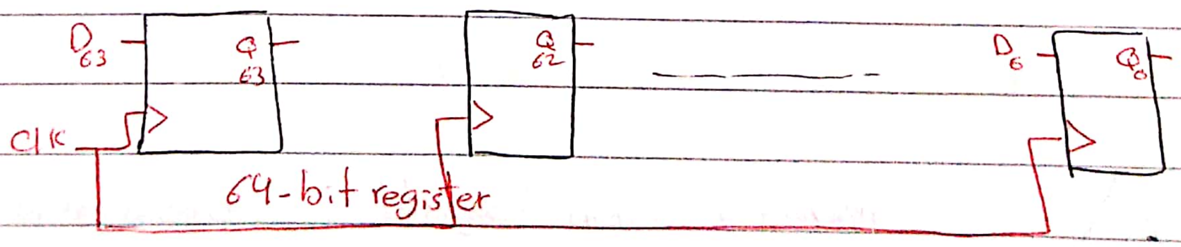
* لازم في الـ sub يكون بالترتيب الـ operant.

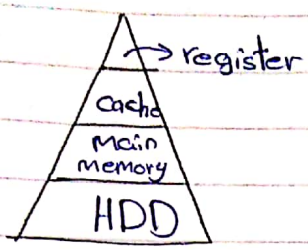
other ex

add f, g, h
Sub f, f, i
Sub f, f, j

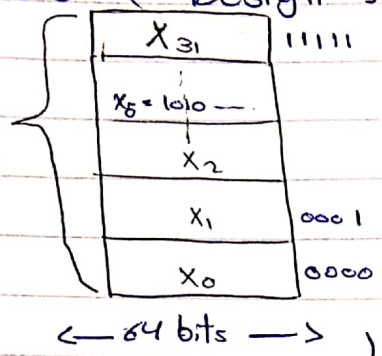
8/10/2019

Ex 8





register file & (Design ال)



* Note &
add c,a,b => add x5 , x7, x5
destination write
Source "read"

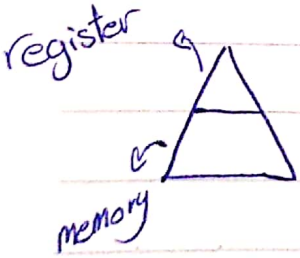
($2^5 = 32$) 5 = bits: عدد ال

* Note & * 8-bit = byte * 64-bit = double word
* 32-bit = word

* يستخدم 32 register لأنه كلما كبر ال register file يصير أبطأ & 32 بكون سريع والعدد مقبول
* كل ما يزيد عدد ال bits يصير أحسن لأنه بيقبل كتير من ال "Instructions"

EX (Page 8 ch 2) & add x_5, x_{20}, x_{21}
add x_6, x_{22}, x_{23}
Sub x_{19}, x_5, x_6

Ex & add x_{19}, x_{20}, x_{21}
Sub x_{19}, x_{19}, x_{22}
Sub x_{19}, x_{19}, x_{23}



* رح افترض إنه أي شي كتت ال "register" بسميه "memory"

No: -------------- Date: --------------

* To apply arithmetic operations :

① Load : reading from memory and writing to the register file (RF).

② Store : reading from RF and writing to the memory.

Ex : int Array-B[10];

"64-bit" : حيكون كل element من هدول ←

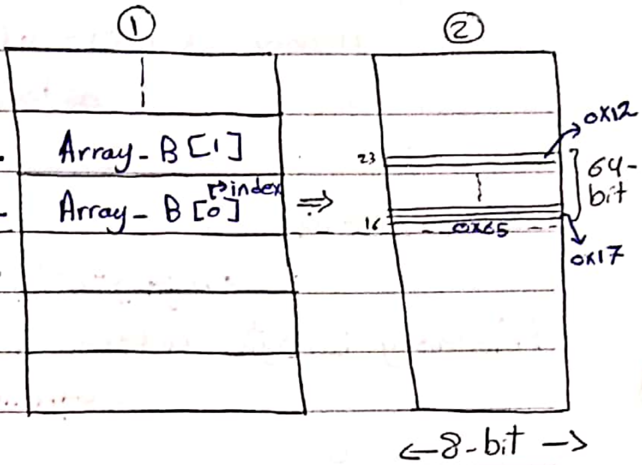
OX1234ABCDFE081765 ←

بتخزن least sig لـ ال

بالأقل وهكذا'

هي الرسمة رقم ② ⇐ الرسمة الأزرق

انظر رسمة ②



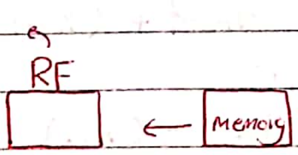
*Important : actual Memory address = base address of the Array + (index * Size)

تطبيق عالمثال اللي فوق : Actual memory address = 16 + (index * 8)
= 16 + 1*8 = 24

لـ هنا الوحيد المتغير الباقي ثوابت

Load (Ld) :

Ex : Ld X_{28} ⇒



* الـ base register ← بيكون مخزن فيه "base address of the Array".

* الـ offset ← بيكون مخزن فيه " index * Size "

Ld X_{28}, 32 (X_{18})

offset ← ↳ base register

(X_{28}) ⟵ memory [32 + (X_{18})] هون في distination

لـ هاي وظيفة الـ CPU

YASSIN

Scanned by CamScanner

No: ---------- Date: ----------

* Store : (Sd) , d X_{28}, 32 (X_{18})

بقرأ منه

Memory [32 + (X_{18})] ⟵ (X_{28})

هون بقرأ الـ address وبتخزن في Memory

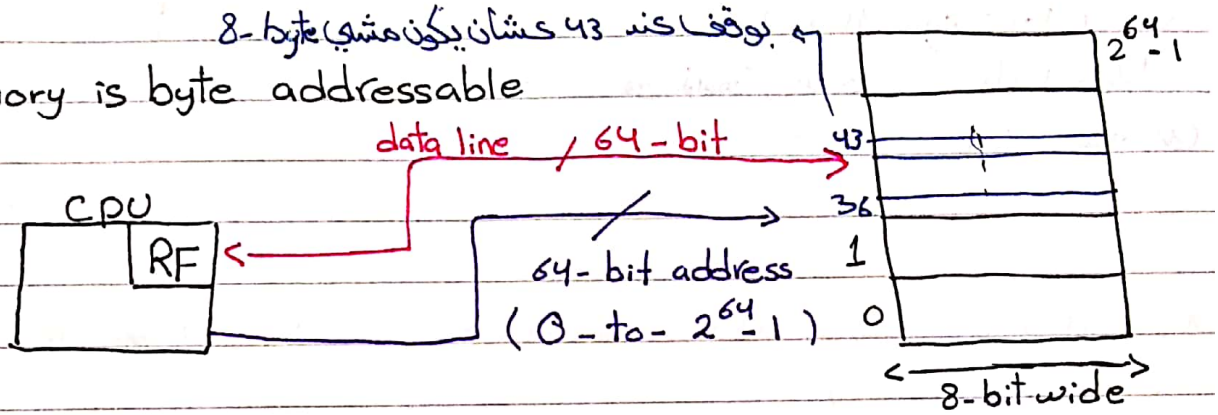
هون ماني distination

---

13 / 10 / 2019

بوقفنا عند 43 عشان يكون مشي 8-byte

Memory is byte addressable



EX : Ld X_{28}, 32 (X_{18})

offset ↵ ↳ base register

بإذا كانت مثلاً 4

cpu ⟶ 32 + (X_{18}) = 36

وبروح بجيبها بال RF عند X_{28}

* (X_{28}) ⟵ Memory [offset + (X_{18})]

EX : Sd X_{28}, 32 (X_{18})

* Memory [offset + (X_{18})] ⟵ (X_{28})

Ex: write an instruction that reads the memory starting from address 8 and write the data into register X_{17} ?

Sol: Ld X_{17}, 8 (X_0)

لأنو X_0 دايما بتبلش من 0

(0).

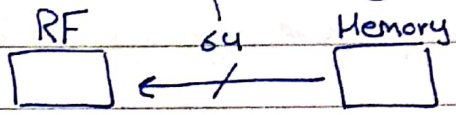
YASSIN

No: ---------- Date: ----------

Ex: g = h + A[8] ← index
x_{20} x_{21} add x_{20}, x_{21},

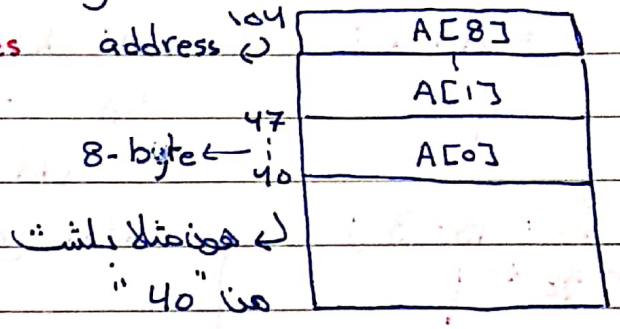
Sol: Ld x_5, 64 (x_{22})
لـ ضربت الـ index بـ 8

لأنه احنا بنتعامل مع double



(load double)

* المكان اللي بلش فيه يسمى
(starting address) / (base address)



* offset = index * Size in bytes
8 = وبنا هنا هو ثابت

هو مثلا بلش من "40"

* Memory address = (base address register) + (index * Size in byte)
offset

Ex: A[12] = h + A[8] page 14

* إذا بشوف Array على اليمين بتكون load ولما تكون الـ Array على اليسار بتكون Store

Sol: Ld x_6, 64 (x_{22})
add x_6, x_6, x_{21}
Sd x_6, 96 (x_{22})

عادي كتبنا هرتين وبقدر اكتب بالـ add x_7 بدل x_6 بس إذا هيك بحط بالـ Sd x_7 . الفرق بهاي الطريقة إني استخدمت (more registers)

Ex: A[12] = h + A[8]
x_{22} ← g = A[8] - h ;

Sol: Ld x_6, 64 (x_{22})
add x_7, x_6, x_{21}
Sd x_7, 96 (x_{22})
Sub x_{22}, x_6, x_{21}

كشان أضفت Sub هاكتبنا الثنتين x_6 كشان x_6 ما تغير واضطر اكتب كمان Ld قبل الـ Sub ، فوجت خليت بالـ add x_7 لأني بحتاج x_6 لتحت وما بغيرها.

No: ..... Date: .....

* إذا بدي أجيب value جديدة أوديها على RF بس هو هلبان بشوف اختار أقل register رح استخدم الـ data تبعته وبضهي فيه ورجعه عالـ memory وبودي الـ value مكانه

* Immediate operands 8

Ex: B = A + 4 ; ⇒ addi x_{21}, x_{20}, 4

هي بتسمحلك تخلي القيمة الثالثة ثابتة. وهي أسهل منا استخدام الـ add لأنك حتضطر تسوي Ld. وبتسمح للثابت يكون + أو -.

D = B - 3 ;

x_8 x_7

addi x_8, x_7, -3

↳ $x_7 + (-3)$

E = A + B + 2 ;

* addi → range بتسمحلك قط ثابت بـ معينة.

* "لو كانت b-3 هون بضطر أعمل Ld"

* The constant Zero 8

Ex: D = 3 - B ; ⇒ Sub x_{10}, x_0, x_9

x_8 x_9

↳ معناته هون

-B خزن

addi x_8, x_{10}, 3

15/10/2019

* Binary integers :

* In any number with base(r), the value of the i^{th} digit (d)

Value = $d \times r^i$

e.x : 0 1 1 0 — 3 2 1 0

value = $1 \times 2^2 = 4$

value = $1 \times 2^1 = 2$

e.x : $X_5 = b_{63} \text{ ----- } b_2 b_1 b_0 \rightarrow \text{--}\,1111 = 2^4 - 1$

$= b_{63} \times 2^{63} + \text{ ---- } + b_1 \times 2^1 + b_0 \times 2^0$

* أكبر قيمة بقدر أمثلها إنه يكون كل الـ bits = 1

e.x : $1111 = 2^4 - 1 = 15$

3 2 1 0

* overflow = إنه الناتج مايكون بوسع عدد الـ bits المتاحة لي (64-bits)

في RISC-V بترك الخيار للمستخدم كيف يتعامل مع الـ overflow، هي ما بتعمل اشي.

* Signed Binary Integers :

الموجب بعبر عنه بـ (0) والسالب بعبر عنه بـ (1)

3 Representaion of Signed number :

① Sign - magnitude → آخر bit إذا كانت 1 ← العدد سالب وإذا 0 ← العدد موجب

e.x : $\underbrace{0}_{+}\underbrace{100}_{4} = +4$

* أصغر عدد فيه = 1111 --- 1 وأكبر عدد = 111 --- 0

② 1's complement → إني أعكس كل bit ونفس الـ Signed بعبر عالموجب بـ (0) والسالب بـ (1)

e.x : 0001 → (+1)

1110 → (-1)

* أكبر رقم = 0111 ⇐ $2^{n-1} - 1$ وأصغر رقم = 1000 ⇐ $-(2^{n-1} - 1)$

③ 2's complement → وهي المستخدمة. وطريقتها بضل ماشيك من اليمين لليسار لحد ما أوصل أول 1 بنزله وبعكس كل اللي قبله. والموجب بعبر عنه بـ (0) والسالب بـ (1)

$b_{n-1} \text{ --- } b_1 b_0 \Rightarrow \text{value} = b_{n-1} \times -2^{n-1} + \text{ ---- } b_1 \times 2^1 + b_0 \times 2^0$

أكبر رقم = $2^{n-1} - 1$ = 0111111 --- وأصغر رقم = 1000000 --- 00 = -2^{n-1}

* 2^{n-1} مش موجود بال 2's complement وهي الوحيدة اللي إلها 0 بتمثيل واحد مش +و-

* (+16) 0 0010000

+ (-24) 1 0011000

1 0101000 →

هيك بال Signed وهذا غلط فصرنا نستخدم 2's complement → كشان يطلع الناتج الصح

* Note : $X + (-X) = 2^n - 1$

* Most Significent bit → هي اللي بتحدد العدد موجب ولا سالب.

e.x : $(1111)_2 = (-1)_{10}$ بال 2's complement

↳ $1\times2^0 + 1\times2^1 + 1\times2^2 + -1\times2^3 = 1+2+4-8 = -1$

E.x :

+16	00010000
-24	11101000
	11111000 → = -8 → 2's comp لما أعمل
	00001000 ↩

24 = 00011000

-24 → 2's comp = 11101000

* الطريقتين لإيجاد الـ 2's complement

① a) 1's complement

b) add 1

② أمشي من اليمين لليسار أول 1 بنزله وبعين بعكس كل اللي بعده.

* $X + \bar{X} = 111\text{---}1 = 2^n - 1 = -1$

$X + \bar{X} = -1 \Rightarrow \boxed{-X = \bar{X} + 1}$

(عملية حسابية) negation ↩ ↳ complement

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

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

$\Rightarrow 11111\text{--}1+1 \Rightarrow 1000\text{--}000 \Rightarrow 2^n$

e.x : -2(1---11111110)64 باستخدام (2's comp)

↳ +2 = 0---00010

* Sign extention : إنّ أكبر الرقم

① unSigned number : بزيد أصفار على اليسار.

e.x : +2 = 0000 0010 => 0000 0000 0000 0010

② Signed number : most Significant bit بزيد حسب ال على اليسار.

e.x : +2 = 0000 0010 => 0000 0000 0000 0010

-2 = 1111 1111 1110 => 1111 1111 1111 1110

* Note : add / sub

load / Store $x_?$, offset ($x_?$)

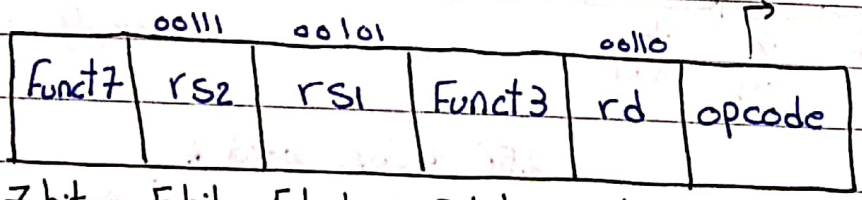
addi 64-bits ← $x_5, x_{10}, 5 \rightarrow$ 12 bits

Signed extension → لما اجمع ومختلفة كتير عدد ال bits بعمل الشرح مفصل بعدين

2.6 : representing Instructions :

* 32-bits كلشي بكون
* Formats مونكين حسب
* Regularity → منتظم (قريبة)

R-Format : register_format



بساعدك على تحديد نوع ال instruction

7 bits 5 bits 5 bits 3 bits 5 bits 7 bits → 32 bits ثابت دائما =

e.x : add x_6, x_5, x_7 (rd rs_1 rs_2)

رقم ال destination → لـ هون بتخزن و x_6 = 00110

إنه يعطيني مساحة اختيارية.

2^7 = 128 instruction

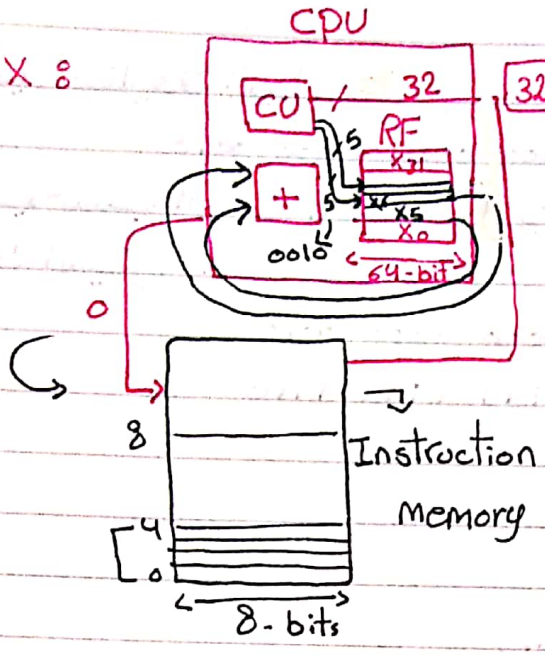
opcode + Funct 3 + Funct 7 بس

2^{17} instruction

Date:

17/ 10/ 2019

E.X :



32-bit add x_7, x_5, x_6

* Very Important Note *

64-bit <= data الـ *

32-bit <= instruction الـ *

=> 32-bit add x_7, x_5, x_6 (rd, rs_1, rs_2)

Funct7	00110	00101	Funct3	00111	opcode 7-bit

→ "هنا هو الـ machine code"

* e.x 8 $(015A04B3)_{16}$ →

0000 0001 0101 | 0100 0000 | 0100 | 1011 | 0011

x_{21} x_{20} x_9 add

بدي أخلي الـ code Assemply

يحول من 16 إلك Binary ومنه

Assemply إلك

256 double word *

* تقريبا

2047 byte

2048 byte

* RISC-V I-format example :

immediate	rs_1	Funct 3	rd	opcode
12-bit	5-bit	3-bit	5-bit	7-bit

64-bit 12-bit

addi rd, rs_1, imm ① <= بيستخدم في الـ

Ld rd, offset(rs_1) ②

12-bit 64-bit

offset / imm → extension بعملها <=

* Note : 2's complement

0 111 — — — 1 = + (2^{11}-1) => + 2047   أكبر نطاق ←
(11 / 210)

1 000 — — 00 = -2^{11} = - 2048   أقل نطاق ←

* RISC-V S-format Instructions : تستخدم في Store instructions
← sd rs₂, offset (rs₁)

imm [11:5]	rs2	rs1	Funct 3	imm [4:0]	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

* المقارنة : جميعهم في نفس المكان بينما الـ imm تتغير مكانه
↓

imm [11:5] 7-bit	rs2 5-bit	rs1 5-bit	Funct 3	imm [4:0]	opcode 7-bits	S
Funct 7	rs_2	rs_1	Funct 3	rd	opcode	R
imm [11:0]	rs_1	Funct 3	rd	opcode		I

Regularity ← الوظيفة => تخليهم

index * Size = 8

ex Page 35 :

```
Ld   x9, 240 (x10)
add  x9, x21, x9
addi x9, x9, 1
Sd   x9, 240 (x10)
```

* Note: Binary compatibility معناها (انه الـ standard تكون موحدة)

No:-------------- Date:-------

20/10/2019

* logical operations Page 37.

* Shift Immediate operations : Page 38

* Shift left logical "Slli"
* Shift right logical "Srli"

Ex: ① Slli rd, rs1, imm   0100 <= 0010 ← 0
② Srli rd, rs1, imm   0001 <= 0 → 0010

Ex: Slli x_6, x_5, 10

64-bits

x_5 = | 0 ——— 000010 | ← 00000000000

لـ رح يتخزن بـ x_6 فـ x_5 ما بتتغير قيمته

x_6 = | 0 ----- 01 0000000000 | 2048

2^{11} ↓ ↓ 2^{10}

* اسمهم logical لأنه الأرقام بعتبرهم "unsigned".
* أكبر شي بقدر أسويله Shift بدون ما أصفر الـ register هو (63 bits)
لأنه لو 64-bits رح يتصفر. <= أكبر قيمة Shift = 111111 (63)
أقل قيمة Shift = 000000 (0)

* هناك طرق لتصفير الرقم بدون Shift لـ 64-bits مثل : * add x, x0, x0
* أطرح من حاله

* الـ Shift left , إذا بدنا ما يأثر عالاشارة لازم يكون الرقم صغير لأنه ممكن تتغير الإشارة
*هيك بتتغير : Slli | 10000000 | ← 0
*هيك ما بتأثر : Slli | 111111 --- 0 | ← 0
* بالـ Shift right , إذا كان Positive ما بتأثر اشارته أما إذا كان negative حطول بتحول
* Srli 0→ | 0 | * Srli 0→ | 1 |
* أما إذا كان الرقم unsigned عادي استخدم Shift right زي ما بدك .

Scanned by CamScanner

No:---------- Date:----------

* Page 41 (ori)

0x FFF
↓
hexadecimal ↳ 1111 1111 1111 ⇒ (I-format) حنستخدم

Xori Xa, X10, 0x FFF

↳ (1) حيكونناتجها كلها

or ↳ عادي ↳ هون Source واحد بس
↳ immediate : 0x FFF

and ↳ عادي
↳ immediate : 1111 1111 1111 ← 0xFFF

* 1 ⊕ 1 = 0 1 ⊕ 0 = 1
0 ⊕ 0 = 0 0 ⊕ 1 = 1

XoR → I-Format

Ex : write a code that store 2's complement of register X7 in X5 :

«هناك طريقتين»

⇒ ① Xori X5, X7, 0x FFF ↳ complement
addi X5, X5, 1 ↳ negation

⇒ ② sub X5, X0, X7 ↳ negation

على بعضها أمر بمرحلتين

* Conditional operations : *branch equal (beq) *branch not equal (bne)
* Branch to a labeled instruction if a condition is true.
(تفرع) (control) (تغيير في ال

ex :
branch equal (beq) ⇒ beq rs1, rs2, Lable
if (A == B) ← هاي كانوا ↓ L1 بترجعلا
Exit ⇒ حسب
loop المطلوب

ex page 43 :

f, g, h, i, j
↓ ↓ ↓ ↓ ↓
X19 X20 X21 X22 X23

beq X22, X23, IF → true انا كانت
Sub X19, X20, X21 ↳

انت سميتي زي مابدك ↳
IF : add X19, X20, X21
Exit : - — —

هاي اذا طلعوا مش مشاوي يعني هاي كانوا الـ "else"
beq X0, X0, Exit ↳
يعني اطلع بعد ما تنفذ الـ branch

YASSIN

No: ---------- Date: ----------

* (Code ثاني)

bne (branch not equal) X_{22}, X_{23}, Else
add X_{19}, X_{20}, X_{21}
beq X_0, X_0, Exit
Else: sub X_{19}, X_{20}, X_{21}
Exit: - - -.

EX 8 (مهم) page 44

while (Save[i] == k) i += 1 ;
Sol 8 كأنهاهيك → while (save[i] == k) i++;
i ≡ X_{22} → register هاي في
k ≡ X_{24} → register وهاي في
base address of array X_{25}

هون أنا بعمل load لـ k == save[i] لأنه حتى لو كانت حعمل نفس الشي لأني حربقارن & عشان تخزين بس check

(8 * i + 25) بدي أعلاهيك

Loop: Slli X_{10}, X_{22}, 3

add X_{10}, X_{10}, X_{25}
Ld X_9, 0 (X_{10})
↳ علت كل هذا عشان أخلي هذا constant

bne X_9, X_{24}, Exist
addi X_{22}, X_{22}, +1 → بس هيك بنكب
beq X_0, X_0, Loop

Exit: - - -.

أو
إذا بدي استخدم beq بس
beq X_9, X_{24}, cont
beq X_0, X_0, Exit
cont: addi X_{22}, X_{22}, +1
beq X_0, X_0, loop
Exit: - - -

No:............ Date:.............

27/10/2019 .....

Four instructions :

blt : branch if lessthan => e.x a > b

bge : branch if greater than equal => e.x a <= b

beq : branch equal ← مناهون بنفرديها

bne : branch not equal

EX : Page 45 :

blt x_{23} , x_{22} , L1

beq x_0, x_0, Exit → لازم يكون بينهم اشي عشان مايصير ال Condition على الفاضي.

L1: addi x_{22}, x_{22} , +1 → مارح ننفذ إلا إذا كان a > b

Exit: ---- --

or :

a > b => a <= b (دكسها)

bge x_{23}, x_{22}, Exit

addi x_{22}, x_{22}, +1 → مارح ننفذ إلا إذا كان a > b

Exit: ______

* Note : x_5 : 11111 00 --- 0 ⌉ → في فرق إذا اعتبرناهم unsigned

x_{16} : 01110000 00 -- 0 ⌋ أو signed وأنا بقارن.

bgeu, bltu ← وإذا بدي أخلي unsigned بضيف حرف u بالأخير

* Bounds check shortcut : → هون ال compiler بس بيتأكد

e.x : Size = 10   A [j]   انه ال j بين

↳ 0 → 9   0 ≤ j < Size → x_{11} ال Size

x_{20} ↲   اللي هو 0-9

bge x_{20}, x_0, L1 بينهم

L1 : blt x_{20} , x_{11} , L2 → beq x_0, x_0,   out of bounds

L2 : تكملة ال code تبعي بينهم → beq x_0, x_0,   out of bounds

or

bgeu x_{20}, x_{11}, out of bounds → عشان اختبر أول شرط j ≥ 0

أنا ضامنة انه موجب ↲ →

YASSIN

No: Date:

لو كانت j ← Positive أو negative رح يقدر يقارن باستخدام

ال instruction ← bgeu x20, x11, out of bounds ← لأنه بتعامل مع الـ unsigned

ex: x_{20} = 1 - -] → هون رح يعتبر x_{20} موجب وأكبر
x_{11} = 0 - - لأنه بتعامل دايماً unsigned
ل دايما موجبة

ففي كلا الحالتين حياخدك على ال out of bounds

* compiling Loop statement :
  * For loop
  * case / Switch

ex : i = x5 , base address of save in x20

Page 43

* memory address = base + offset
  = base + index size
  = base + index × 8 → = 2^3

```
      add   x5, x0, x0
      add   x6, x0, +10
Loop: bge   x5, x6, Exit
      slli  x7, x5, 3
      add   x7, x20, x7
      ld    x9, 0(x7)       → zero
      slli  x9, x9, 1
      sd    x9, 0(x7)
              zero ↵
      addi  x5, x5, 1
      beq   x0, x0, Loop
Exit: ______
```

No: ---------- Date: ----------

e.x : Switch (n) ↳ x_{20}

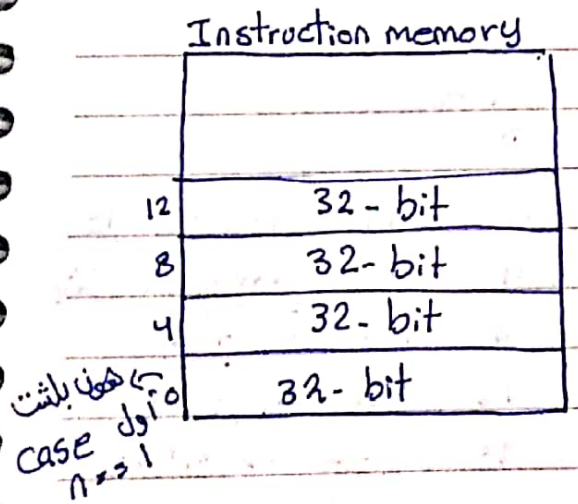
```
case 1: ____
  break;
case 2: ____
  break;
  :
default: ____
  break;
```

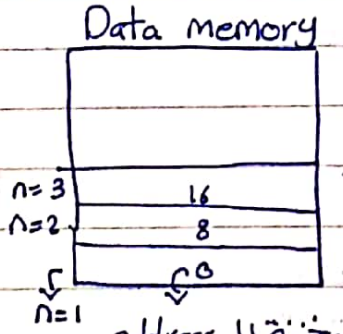
```
if (n == 1)
  _____;
else if (n == 2)
  _____;
  :
else _____;
```

```
addi x5, x0, 1
bne x20, x5, L2
  ____;
beq x0, x0, Exit   → break بدل
L2: addi x5, x0, 2
bne x20, x5, Else
  ____;
beq x0, x0, Exit
Else: _____;
Exit: _____;
```

هون بحدد ال Instructions جد عالي فهناك طريقة أسهل.

÷ لو عنا switch ← 100 رح نزيد ← 3*100 = 300 instructions





مخزنة ال address

بروح عند ال Memory منفذ ال address تبع 0 وهكذا

YASSIN

No: .... Date: ....

29/10/2019

Slide 49: beq rs1, rs2, label

↓ label: address of the target instructions

No branch targets ← address of the target instructions

بس اول اشي بدمين

2.8 : Slide 50 :

EX :

```
int sum (int a, int b)
{
    return a+b ;   -> a,b -> بنخزنوا
}                            ب register
```

Sum ② transfer →

```
int sum(int a, int b)   ① place Parameters ...
{
    int total ;         -> ③ acquire Storage
    total = a+b ;
    return total ;      -> ⑥ return
}
```

* Slide 51 : jump and link (jal)

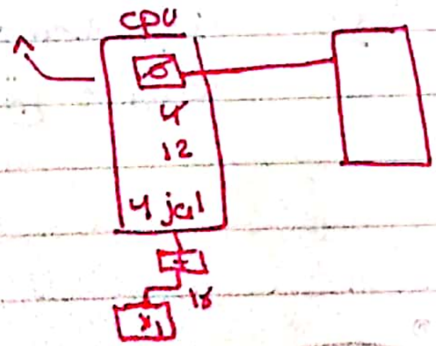
jal rd1 label
↓
x_1

address
↑

④ (

address		
0	add	return عشان نعمل
4	Sub	بنخزن ال address
8	ld	+4
12	jal	x_1, Sum
16	---	$x_1 = 16$

Program counter (PC)
القيمة تاعت ال address
PC+4



No: ---------- Date: ----------

* jump and link register (jal r)

jal rd , offset (rs1)

① (rd) ← pc + 4
of " jal r "

② (pc) ← offset + (rs1)

ex:
0 add
4 sub
8 Ld
12 jal x , sum
16 . . . . .

Sum ------ * هاي متزنين
فيها 16 و offset = 0

16
16 فيرجع لـ

Pc jal r x0 , 0 (x1)
offset

* unconditional branch * الشرط دائما متحقق

« بعادلوا بهذ »

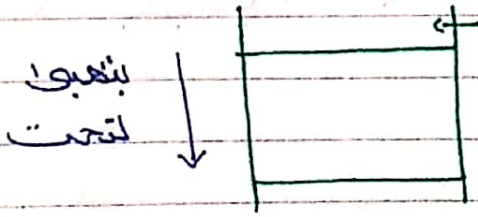
beq x0 , x0 , Exit ≡ jal x0 , Exit

Slide 52 :

Stack : last in First out " LIFO "

push → store

pop → load



* أطمكان بال stack بكون x2 دايما

* إذا احتجنا نقرأ اشي من ال Stack من Stack pointer

ex: Push

sd rs2 , offset (x2)
sp

pop ← Ld (sp)

YASSIN

Slide 53 :

leaf

* اسمه leaf لأنه ما بنادي واحد بعده ← procedure

EX :

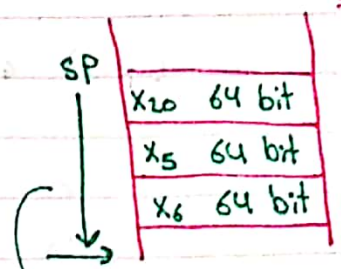
24= 3*8 : فخزنهم ف ← ③ لأنه نزل ←

Push

x_{20}, x_5, x_6

```
addi  sp, sp, -24
sd    x20, 16(sp)
sd    x5, 8(sp)
sd    x6, 0(sp)
```

*



operation

```
add   x5, x10, x11
add   x6, x12, x13
Sub   x20, x5, x6
```

(x20, x5, x6) لأنه قيمهم ما بتتغير

Pop

```
addi x10, x20, 0
Ld   x20, 16(sp)
Ld   x5, 8(sp)
Ld   x6, 0(sp)
addi sp, sp, 24   → Pointer هيك برجع الـ لمكانه الأصلي
jalr x0, 0(x1)
```

* هاي داعي نعمل Store و load لـ temporaries بس الـ Save .

* بصير هيك ←

```
addi  sp, sp, -8
Sd    x20, 0(sp)
add   x5, x10, x11
add   x6, x12, x13
Sub   x20, x5, x6
addi  x10, x20, 0
Ld    x20, 0(sp)
addi  sp, sp, 8
jalr  x0, 0(x1)
```

YASSIN

No: ..... Date: .....

Main :

x_{10} x_{11} x_{12}

z = leaf example (7, 5, 3, 6);

z = 12 - 9

z = 3

```
0    addi   x10, x0, 7
4    addi   x11, x0, 5
8    addi   x12, x0, 3
12   addi   x13, x0, 6
16   jal    x1, leaf example
```

31 / 10 / 2019

ex: n = 5

fact (5) = 5 x 4 x 3 x 2 x 1 = 120

fact (1) = 1

fact (0) = 1

n = 5 → 5 x Fact (4) → 24 x 5 = 120 (call)

4 x Fact (3) → 4 x 6 = 24 (call)

3 x Fact (2) → 3 x 2 = 6

2 x Fact (1) → 2 x 1 = 2

1 x Fact (0) → 1 x 1 = 1

No: Date:

3/11/2019

Page 58 :

```
20 Fact : addi sp, sp, -16
24 Sd   x1, 0 (sp)
28 Sd   x10, 8 (sp)
32 addi x5, x0, 1
36 bge  x10, x5, Else
40 [ addi sp, sp, 16 ]
44 addi x10, x0, 1
48 jalr x0, 0 (x1)
```

إذا علت هون ld عالفاضي ← (40)

لأنه قيم x_{10}, x_5 ما تغيرت نفس ماهي . (44, 48)

* x_{10} x_5

$n \geq 1$

$n-1 \geq 0$

قبل ما أعمل return لازم أرجع كل Value بال stack زي ما كانت

Else : jal x1, fact X

x_1 لـ دايماً خليه

حنعمله هيك مش زي فوق

```
52 Else: addi x10, x10, -1   ✓
56 jal  x1, Fact
60 addi x6, x10, 0  →  x6 = Fact (n-1)
64 Ld   x1, 0(sp)       x10 = Fact (n-1)
68 Ld   x10, 8 (sp)
72 addi sp, sp, 16
76 Mul  x10, x6, x10
80 jalr x0, 0 (x1)
```

قيمة n باخذها من ال Stack

دايماً برجع لـ 60

Main :

```
0 addi x10, x0, 4
4 jal  x1, Fact
8 addi x20, x10, 0
  ⋮
```

$x_1 = 8$

$x_{10} = 24$

$x_6 = 6$

⇓

اخر قيم لـ x_1, x_{10} و x_6

Stack
4
8
3
60
2
60
1
60
0
60

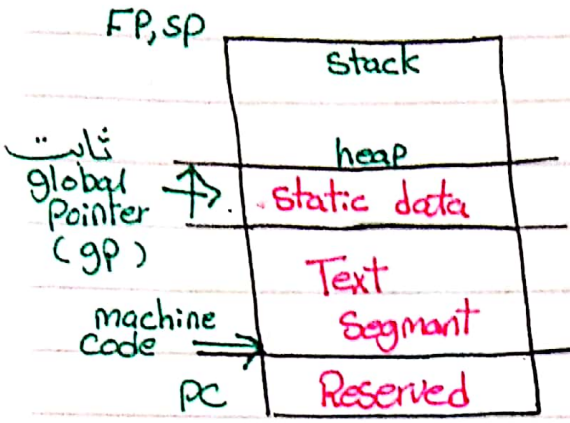
* Memory layout slide 61 &

* لما أحط الـ gp بالنص كشان (أكبر) لأنه (+) و (-)

* Ld بـ (gp^{x_2})

↳ $(-2)^{11}$ to $+(2^{11}-1)$

*

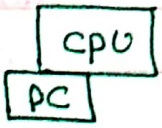


Vistual Address space

↓

16 kB - 64 kB مقسم تقريبا من

وفي الـ Memory بخزن الجزء اللي في الاشي اللي بدي ياه.



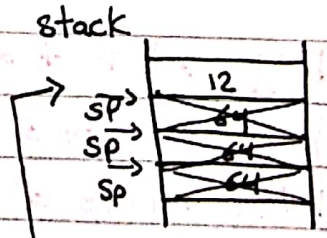
5/11/2019

Slide 62 & Non leaf → لازم اخزن الـ x_1

```
40 Sum : addi sp, sp, -8
     44 Sd   x1, 0 (sp)
     48 bge  x0, x10, Else
     52 add  x11, x11, x10
     56 addi x10, x10, -1
     60 jal  x1, Sum
     64 beq  x0, x0, Exit
Else : add x12, x11, x0
```

الترتيب مهم لإنه بجمع بعدين بطرح

```
Exit : Ld x1, 0 (sp)
       addi sp, sp, 8
       jalr x0, 0 (x1)
```



Ex على فوق & main :

```
0  addi x10, x0, 3
4  addi x11, x0, 0
8  jal  x1, Sum
12 addi x20, x12, 0
```

* اخر قيم لـ &:

$x_{10} = 0$

$x_{11} = 6$

$x_1 = 64$

$x_{12} = 6$ →

YASSIN

No: Date:

[OR] iteration

```
long long int sum (long long int n, long long int acc)

while (n>0)
{
    acc = acc + n;
    n = n - 1;

}
  return acc;
```

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

→ 2 codes

* نلاحظ الفرق بين الـ
هنا أقل كلفة وأحسن *

* Slide 65 :
* Sign extend to 64 bits in rd :
* lb ≡ load byte (8-bit)
* lh ≡ load half word (16-bit)
* lw ≡ load word (32-bit)
* ld ≡ load double word (64-bit)

[lb] → byte of 32
[lh] → byte of 32, 33
[lw] → byte of 32, 33, 34, 35
[ld] → byte of 32, ----, 39

⇒ rd, offset (rs1)

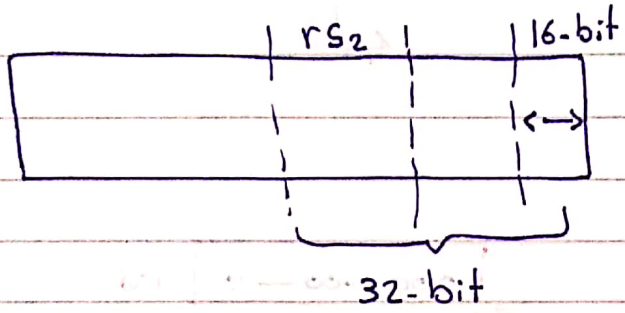
فرض 32 (x0) = 32

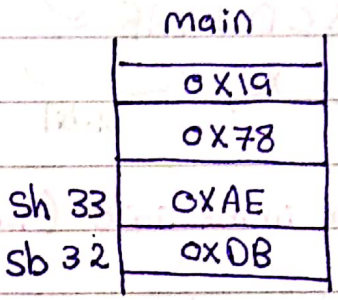
* unsigned : Zero extend to 64 bits in rd
  * lbu
  * lhu
  * lwu
* store byte:
  * sb
  * sh
  * sw
  * sd

rs2, offset (rs1)

32 (x_0)

0XFFBC2345197 8AEDB





Slide 66 : 1000, 000, 000 → 64-bit

16 x 8 = 128-bit → 40-bit

11 x 8 = 88-bit

7/11/2019

Slide 67 :

```
strcpy : addi sp, sp, -8
         sd   x19, 0(sp)
         addi x19, x0, 0
loop:    add  x5, x19, x11
         lbu  x5, 0(x5)
         add  x6, x19, x10
         sb   x5, 0(x6)
         beq  x5, x0, Exit
         addi x19, x19, 1
         jal  x0, loop
Exit: ld x19, 0(sp)
      addi sp, sp, 8
      jalr x0, 0(x1)
```

* Y[i] → i + base address

* x[i] → i + base address

x_5 = 0X 0000 0000 0000 0067 → لو افترضنا إنه

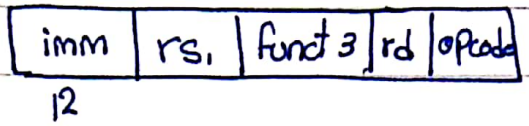
عدد المرات ←

* لما نستخدم char ⇐ الأفضل نتعامل مع "unsigned"

* خلص "Slide 80" من ch2

* 32 - bit constants

* addi



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

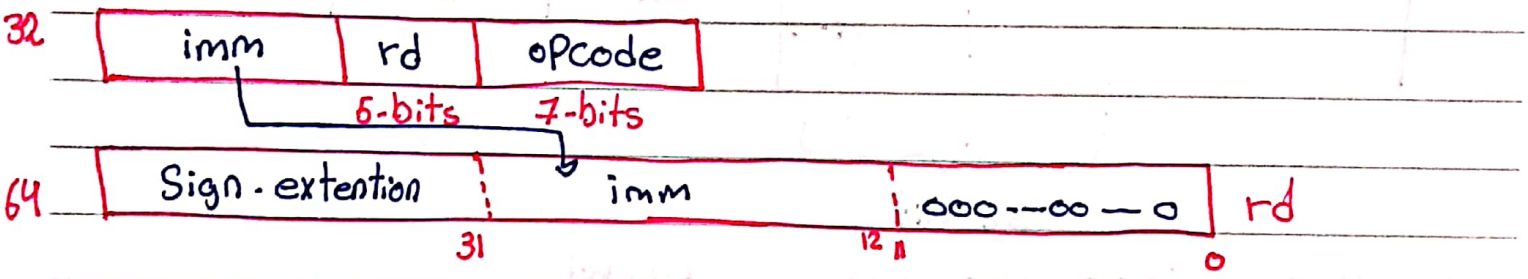
x_{20}

$X = 0X\ 73CD\ 14AF$ ← 32-bits

addi x_{20}, x_0, ← 12-bit

بنفش أحطها بس فوق الـ bits فالحل استخدام

* load upper immediate (lui)



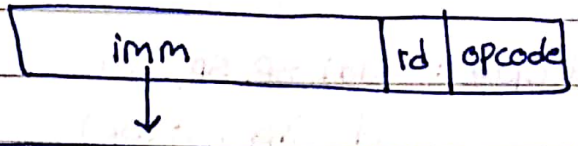
X = 0X 73 CD 14 AF

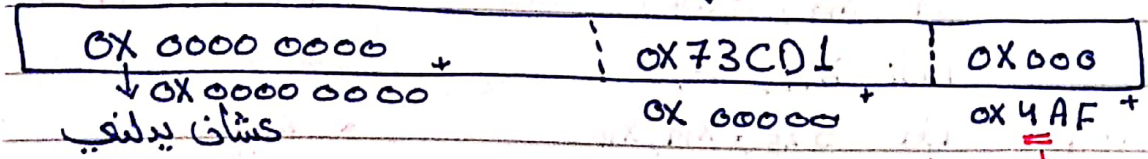
* لحل المثال في الأعلى:

Sol: lui x_{20}, 0X73CD1

addi x_{20}, x_{20}, 0X4AF

x_{20} = 0X 0000 0000 73CD14AF





كشان يدلني hexadecimal انه,

" 0 الها Sign extention " الـ ⇐

ex 8 X_{20} : X = 0X 73CD 1 8 AF (32-bit)

1 = Sign extention هون ال

Sol:

0X 0000 0000	0X 73CD1	0X000
0X FFFF FFFF	0X FFFFF	0X 8AF

⇒ lui X_{20}, 0X 73CD2 → لأنو الناتج طلع معي غلط فعشان اخليه صح بعمل CD2

addi X_{20}, X_{20}, 0X 8AF

X_{20} = 0X 0000 0000 73CD18AF

Ex : lui X_{10}, 0X ABCDE « سؤال بالعكس »

addi X_{10}, X_{10}, 0X 825

AFter this code. what is the value of X_{10}

Sol:

X_{10} = 0X FFFF FFFF ABCDD825 (16-bit)

* تفصيل الحل * :

X_{10} =	0X FFFFFFFF	ABCDE	0X000
	0X FFFFFFFF	FFFFF	0X825 +

* Branch Addressing
  * Format of Branch :

imm[12] imm[10:5]	rs_2	rs_1	Funct3	imm[4:1] imm[11]	opcode	SB-Format
7-bit	5-bits	5-bits	3-bits	5-bit	7-bit	

7-bit	rs_2	rs_1	Funct3	5-bit	opcode	S-Format
imm[11:5]				imm[4:0]		

• الاختلاف اللي بينهم بسيط جدا •

ex& * توضيح ليه ضربنا ب 2 *

0 loop: ____
4 ____
8 ____
12 ____
16 beq x_0, x_0, -8

* 16 + -8 x 2 = 0
لـ loop اللي هي ال

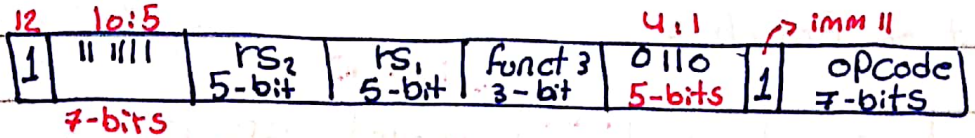
-8 & عدد ال inst بين ال branch وال loop بدون ال branch
مضروب x 2
=> 4 x 2 = 8
والسالب لأنه ال branch بالأسفل وال target فوق ال branch

* jal rd, lable
* branch rs_1, rs_2, lable (imm)

* Target address = pc of branch + immediate x 2

Ex& 0 loop: ____
4 ____
8 ____
12 ____
16 ____
20 beq x_0, x_0, loop

Target address = pc of branch + immediate x 2
0 = 20 + immediate x 2 0000 0000 1010
imm = -10 $(-10)_{10} = (1111\ 1111\ 0110)_2$



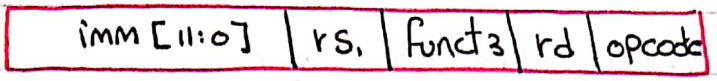
* The immediate represents the offset in halfwords
2 halfwords = instruction لـ unit
كشان هيك بنضرب بـ "2"

YASSIN

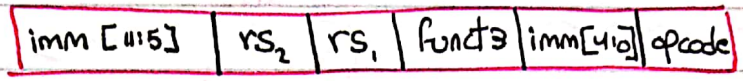
No: ---------- Date: ----/----/----

* مقارنة *

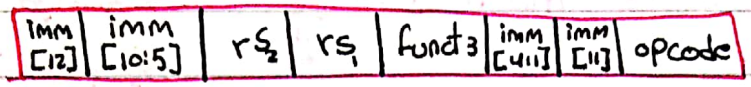
addi , Ld



Sd



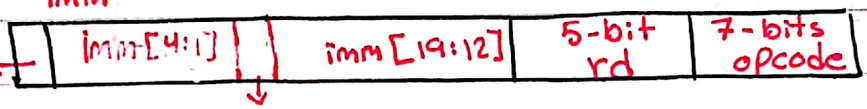
branch



↳ من طلي كمثل الـ Sign * bit

* jal rd, lable ؟

imm



imm [20]

imm [11]

Target = pc of the jal + imm x 2

* branch => -2^{11} to $(2^{11}-1)$ halfwords

-2048 to 2047 halfwords -> -4096 to 4094

± 4k bytes

± k instruction

* jal -> -2^{19} to $(2^{19}-1)$ halfwords

- 512 k to 512 k

Slid 74 ؟ ⊛ 80024 = 80012 + imm x 2
(Exit)

Exit = 6

⊛ 80000 = 80020 + imm x 2

imm = -10

* jalr rd, offset (rs1) → ماحكينا عنه لأنه ماعنده lablé مابعتمد
ع lable

Target address = offset + (rs1)
64-bits

EX: ↘

* 0X 0000 0000 ABCD 123 0 ← 4, 8, C
الي مسموح تحطهم غير 0
↓ 00

lui x_{20}, 0X ABCD1
addi x_{20}, x_{20}, 0X230
jalr x_0, 0 (x_{20})

or ⟹

lui x_{20}, 0XABCD1
jalr x_0, 0X230 (x_{20})

* Note: 1230

← إذا كانت 8 فما فوق بنضيف 1 لـ

lui x_{20}, 0XABCD1

EX: beq x_{10}, x_0, L_1
⇓
bne x_{10}, x_0, L2
jal x_0, L_1
L2: - ——

± 4 kB

± 1 MB
بعيدة

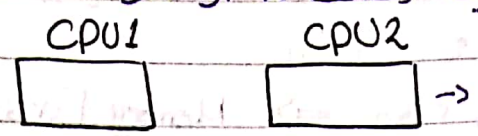
* Slide 77 ⟶ (shift) ال مهمة تكون معك بالامتحان جدًا وضيف عليها

No: ---------- Date: ----------

10/11/2019

* Synchronization :

هذول سبحولك أيضاً إنك تنفذ الـ code اللي انت فيه بس ومابحيس تضارب بين 2 codes أو أكثر



lock();

```
Ld X19, 64(X0)
addi X19, X19, 1
Sd X19, 64(X0)
```

*

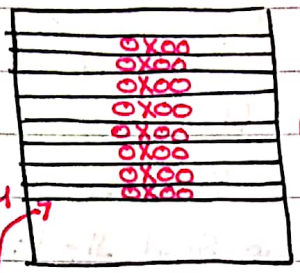
unlock();

lock();

```
Ld X19, 64(X0)
addi X19, X19, 1
Sd X19, 64(X0)
```

*

unlock();



* المفروض تطبق أول وحدة و ترجعلي 0x01 بعدين أنفذ الثانية وترجعلي 0x11 (2) / ولكن اذا أصبح تنفيذ العمليتين في نفس الوقت سيصبح الناتج 0x01 وهذا سيحل باستخدام الـ Synchronization

* Mutual Exclusive Region ("ME" region) وهذا معناه إنه تنفذ ما تسمح بتنفيذ 2 codes أو أكثر بنفس الوقت code - code

* عملية الـ check من الـ Value بتبين إذا الـ lock بيجي الـ cpu بشوف إذا الـ lock (Atomic) Single

* lock , unlock :

Value 0 → lock is free

Value 1 → lock is busy

إذا كانت 0 بخليها 1 وهيك صارت lock واي ولحد بيجي بشوفه 1 فهيك busy

* طريقة لكتابة الـ lock *

lock(); =>

```
loop: addi X6, X0, 1
      Swap X6, lock
      bne X6, X0, loop
```

* بكتبها قبل ما أبلش الـ code الـ 2 codes زي اللي فوق ↑ عشان يزبط إنه ما يدخل على الـ 2 codes بنفس الوقت

* بس هذا الحل ↑ غير مستخدم في الـ Risc-v لإنه بخرافف الوقت واستبدل بطريقة أخرى :

* load reserved : lr.d rd, (rs1)

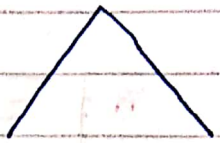
* Store conditional : sc.d rd, (rs1), rs2

* load reserved :
lr.d rd, (rs1) ⟹ (rd) ← Memory [(rs1)]

* Store conditional :
Sc.d rd, (rs1), rs2 ⇒ Memory [(rs1)] ← (rs2)

* Memory

address	Processer
(rs1)	P1
(rs1)	P2



(rd) = 0 Success

(rd) != 0 (fail)

* بكون فيه processor تاني غير ال lr.d وال Sc.d مستخدمه

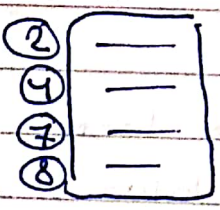
* Swap : (X20)
X23

* الطريقة الخاطئة * : Swap X23, (X20) X

* الطريقة الصحيحة * :

loop : ① lr.d X10, (X20)
③ Sc.d X11, (X20), X23
⑤ bne X11, X0, loop
⑥ addi X23, X10, 0

atomic ⟹ ✓ / Fail



address	Processor
(X20)	P1
X20	P2

← لما جي يعمل ③ رح يلاقي تنتين وهيك ما بربط فبحذف اللي من نفس المكان بشيل P1 X20 هيك لما يجي لـ ④ صارت تربط لإنه بس وحدة P2 X20

No: ---------- Date: ----------

EX مثال:

address of variable lock is in (x_{20})

lock ();

Ld x_{19}, 64(x_0)
addi x_{19}, x_{19}, 1
Sd x_{19}, 64(x_0)

} ME region

unlock ();

addi x_{12}, x_0, 1
loop: Lr.d x_{10}, (x_{20}) < 0 / 1
Sc.d x_{11}, (x_{20}), x_{12}
bne x_{11}, x_0, loop
bne x_{10}, x_0, loop

→ lock

بزبط هيك لانه هيك بشيك أول اذا طلعت مش ناجحة خلص.

unlock عالته ← Sd x_0, 0(x_{20})

* في Slide 80 الـ code بختلف شوي بالترتيب وفائدته بس سرعة في الـ "code".