Chapter 1 — Understanding Performance

عبد الرحمن عصام 0175864

① Algorithm ② programing lang, compiler, Architecture ③ prossor & memory system ④ I/O system.

Eight Great ideas

1 Moore's law
↳ resources double every 18-24 month
↳ computer design takes years, the resources can double or quadruple.

2 Abstraction
يعني انزل من level لمستوى
cpu
team فالأساس يعني
كل واحد يشتغل على جزء
تعتبر محكم كل واحد Black Box
↳ increased productivity

3 common case fast
لو كان في حاجة مرة تنفيذ ال inst اسرع
تأثير perfor.

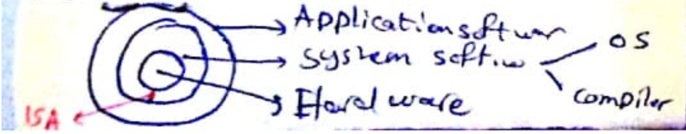
4 parallelism
2 core

5 pipelining
↳ specific type of parallelism

6 prediction
Better ask for forgiveness than ask for permission

7 Hierarchy
→ Caches (fast, small, expensive)
→ HDD (slow, Large, Not exp.)

8 Dependability via Redundancy
① اكثر من comp. تشتغل مع بعض
في حال تعطل الآخر (بدون تسريع)
② نسخة كاملة (نسخ احتياطي) ال 2 components



A five component of all computer

input, output
memory, databath
control
2 Processor

Touch screen
capacitive
Multiple touch
Zoom تكبير وتصغير

BitMap
① size
② Resolution

frame Buffer

(safe place) مكان التخزين

	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
Volatility	Volatile	Non-volatile	Non-volatile
Wearout	N/A	N/A	1,00,000-1,000,000 write

performance

$$\text{Perform.} = \frac{1}{\text{Response Time}}$$

$$\frac{\text{perf}_x}{\text{perf}_y} = \frac{ET_y}{ET_x} = n$$

"x is n time faster than y"

$$\text{cpu perf} = \frac{1}{\text{user cpu time}}$$

$$\text{cpu time} = \text{clock cycles} \times T = \frac{\text{clock cycles}}{\text{clock Rate}}$$

(Rate → freq)

$$\text{clk cycles} = IC \times CPI_{avg}$$

$$\text{cpu time} = IC \times T \times CPI_{avg} = \left(\sum_{i=1}^{n} CPI \times IC_i\right) \times T$$

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

↳ Relative freq (%) بيكون معطي بال

* same computer → "T is the same"
* same compiler + same ISA
↳ (IC_{avg}) & (IC_i) is the same
* same hardware → same CPI_i
But not same CPI_{avg} بيختلف ف IC

ISA → IC
↳ CPI
↳ clk (T_c) fixed

design : CPI

Hardware → clk period → T_c

performance ← يعتمد على → IC, CPI
↳ Prog Lang / algorithm / compiler

RISC-V Reg.

Name	Register number	Usage	Preserved on call?
x0	0	The constant value 0	n.a.
x1 (ra)	1	Return address (link register)	yes
x2 (sp)	2	Stack pointer	yes
x3 (gp)	3	Global pointer	yes
x4 (tp)	4	Thread pointer	yes
x5-x7	5–7	Temporaries	no
x8-x9	8–9	Saved	yes
x10-x17	10–17	Arguments/results	no
x18-x27	18–27	Saved	yes
x28-x31	28–31	Temporaries	no

Example of "Make the Common Case Fast": 12 saved, 7 temporary, and 8 argument is sufficient most of the time

Scanned by CamScanner

```
add  rd, rs1, rs2
sub  rd, rs1, rs2
sll  rd, rs1, const
xor  rd, rs1, rs2
srl  rd, rs1, const
or   rd, rs1, rs2
and  rd, rs1, rs2
lr.d rd, (rs1)
sc.d rd, (rs1), rs2
```
R-type

offset = index x 8

```
lb  rd, offset(rs1)
lh       "
lw       "        signed
ld       "
lbu      "
lhu      "        unsigned
lwu      "
addi rd, rs1, const
srli rd, rs1, const
ori  rd, rs1, const
andi     "
jalr x0, 0(x1)
```
I-type

```
S-type { sb rs2, offset(rs1)
         sh   "
         sw   "
         sd   "
SB     { beq rs1, rs2, L1
         bne   "
         blt   "
         bge   "
         bltu  "
         bgeu  "
U- lui rd, const
UJ- jal x1, Lable
```
32-bit const: lui, addi

Instruction	Format	funct7	rs2	rs1	funct3	rd	opcode
add (add)	R	0000000	reg	reg	000	reg	0110011
sub (sub)	R	0100000	reg	reg	000	reg	0110011

Instruction	Format	immediate	rs1	funct3	rd	opcode
addi (add immediate)	I	constant	reg	000	reg	0010011
ld (load doubleword)	I	address	reg	011	reg	0000011

Instruction	Format	immed-iate	rs2	rs1	funct3	immed-iate	opcode
sd (store doubleword)	S	address	reg	reg	011	address	0100011

beq x10, x0, L1 → bne x10, x0, L2 ; jal x0, L1

Design principle

1) simplicity favor regularity to (achive better perf)

2) smaller is faster
↳ Reg is faster than memo and consum
→ compiler use reg for var
1) to get better perf.
2) conserve energy
3) optimization is imp. Reg

3) Good design demands good compromises
↳ keep format as similar as possible (regular)

signed magnitude: Rang $-(2^{n-1}-1)$ to $+(2^{n-1}-1)$; bit MSD [illegible]

1's comp [illegible]
2's comp [illegible]

slli / add / ld ← 8xi

Basic Block [illegible]

jal ; linking = PC+4 ; x1 ←

↳ addi sp, sp, -[ ]
sd
sd
leaf → saved Reg
Non leaf → saved, regular, argum, temp
ld
ld
addi sp, sp, +[ ]
jalr

Memory layout
1) text, Program code
2) static data x3 → GP static data
3) Dynamic → stack

Char → 1 byte ; 16 byte
"cal" = 4 byte ; 'c' 'a' 'l' '\0' null

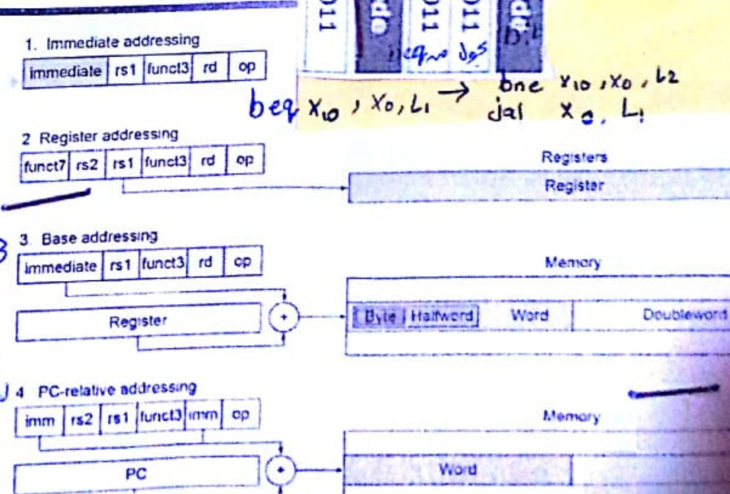
Target Address = PC + imm x2

const 12 → jal ±1 MB
Branch → ±4 KB
lui → 20 B

lui x20, 0x3D0A1
jalr x0, 0x234(x20)
PC [0x3D0A1234]

Example 1: atomic swap (to test/set lock variable)
```
again: lr.d x10,(x20)
       sc.d x11,(x20),x23   // X11 = status
       bne  x11,x0,again    // branch if store fai
       addi x23,x10,0       // X23 = loaded value
```

Example 2: lock
```
       addi x12,x0,1        // copy locked valu
again: lr.d x10,(x20)       // read lock
       bne  x10,x0,again    // check if it is (
       sc.d x11,(x20),x12   // attempt to stor
       bne  x11,x0,again    // branch if fails
Unlock:
       sd   x0,0(x20)       // free lock
```



Scanned by CamScanner