Introducció als Ordinadors:
CAPÍTOL 3
FORMAT DE LES INSTRUCCIONS

CONJUNT D'INSTRUCCIONS

Instruccions: identificadors de les diferents tasques que sap realitzar la CPU.

Cada instrucció té assignat un codi binari únic.

Definirem

instrucció = Codi binari que és capaç de decodificar la unitat de control de la CPU per tal de du a terme una tasca.

Format de la Instrucció forma en que estan codificades les instruccions. Distribució en camps

Format depèn de cada CPU, però bàsicament consisteix en definir una sèrie de camps de cada instrucció,

Cada camp té una situació i longitud determinada.

Camps d'instrucció Tipus d'instrucció

0	6	12	31
Codi d'operació *(opcode)*	**Tipus d'instrucció**	**adreça** (*address field*)	**Mode d'adreça**
l'operació que es realitza	Diferents tipus d'instruccions	On es troben els operants	Mode d'adreçament de memòria

Sovint el opcode i el tipus són un mateix camp

TIPUS D'INSTRUCCIONS

- **Instruccions de Registre**
- **Instruccions de Moviment o de Memòria**
- **Instruccions de Salt Condicional i Incondicional**
- **Instruccions específiques**

INSTRUCCIONS DE REGISTRE (*Register Instructions*)

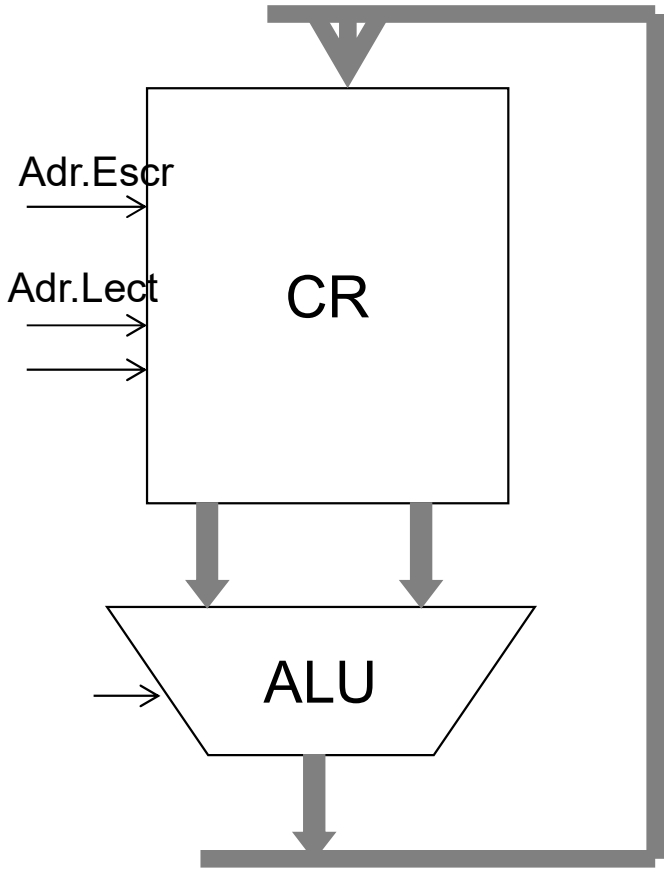
•Operen amb valors guardats a registres
•Realitzen operacions aritmètiques, lògiques sobre operants guardats en els registres de la UP.
•Alguns processadors implementen operacions aritmètiques o lògiques directament amb dades de la memòria

Add RA, RB, RC

equival a:

CR[A] <= CR[B] + CR[C]

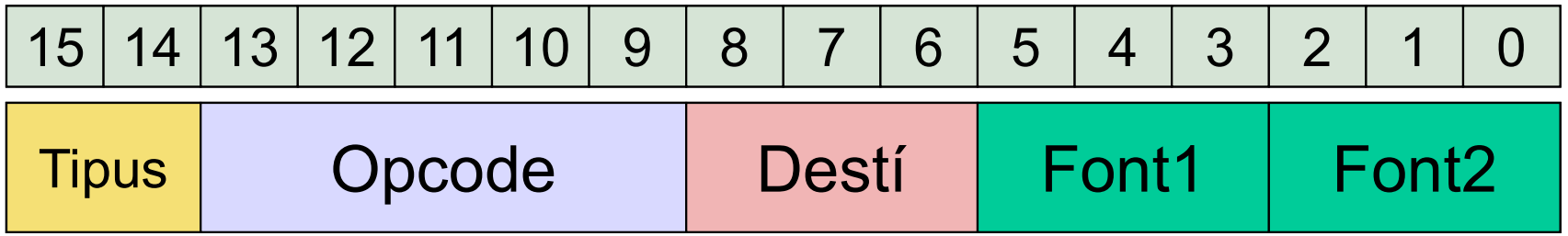
Llegeix continguts dels registres B i C, els suma i el guarda al registre A.



Depenent de l'arquitectura s'haura d'implementar en diversos cicles.

INSTRUCCIONS DE REGISTRE (*Register Instructions*)

exemple



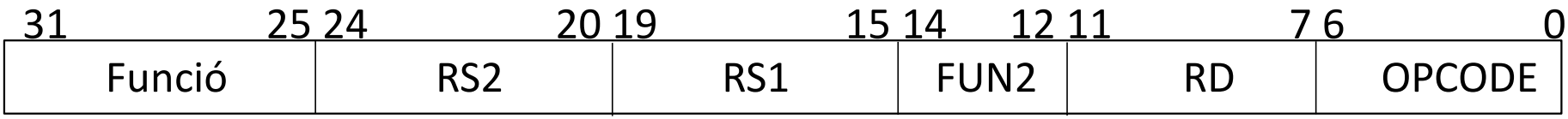
Aritmètica
Lògica
Moure
Desplaçar

Instrucció	**acció**
Op Destí Fon1 Font2	CR[Destí]<=CR[Font1] Op CR[Font2]

Exemple Instruccions de registre

Instruccions Aritmètic – lògiques

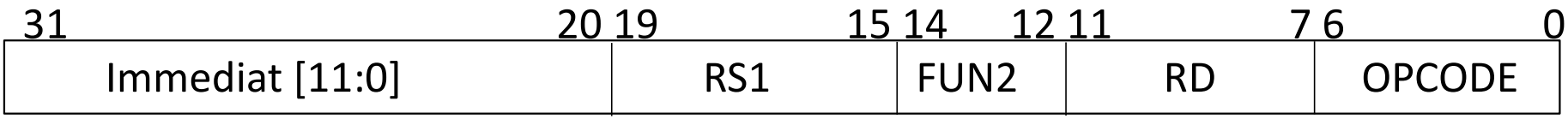
ADD Rd, Rf1, Rf2	Suma el contingut de Registre font 1 + el contingut de R font 2 i ho guarda en R destí
SUB Rd,Rf1,Rf2	Resta el contingut de R font 1 – el contingut de R font 2 i ho guarda en R destí
AND Rd, Rf1, Rf2	Fa una AND dels continguts de Rf1 i Rf2 i ho guarda en Rd
XOR Rd, Rf1, Rf2	Fa una XOR dels continguts de Rf1 i Rf2 i ho guarda en Rd
OR Rd, Rf1, Rf2	Fa una OR dels continguts de Rf1 i Rf2 i ho guarda en Rd



Exemple d'us. Valors Immediats operats amb reg

Instruccions que operen amb immediats

ADDI Rd, Rf1, imm	Suma el contingut de Registre font 1 + el valor immediat imm i ho guarda en R destí
ANDI Rd, Rf1, imm	Fa una AND dels continguts de Rf1 i el valor immediat imm i ho guarda en Rd
XORI Rd, Rf1, Rf2	Fa una XOR dels dels continguts de Rf1 i el valor immediat imm i ho guarda en Rd
ORI Rd, Rf1, imm	Fa una OR dels continguts de Rf1 i el valor immediat imm i ho guarda en Rd
LI rd, imm	Guarda una constant (imm) en Rd



INSTRUCCIONS DE MOVIMENT O DE MEMÒRIA (*Move Instructions*)

Mouen dades entre memòria i registres.
Les instruccions més corrents de moviment són:

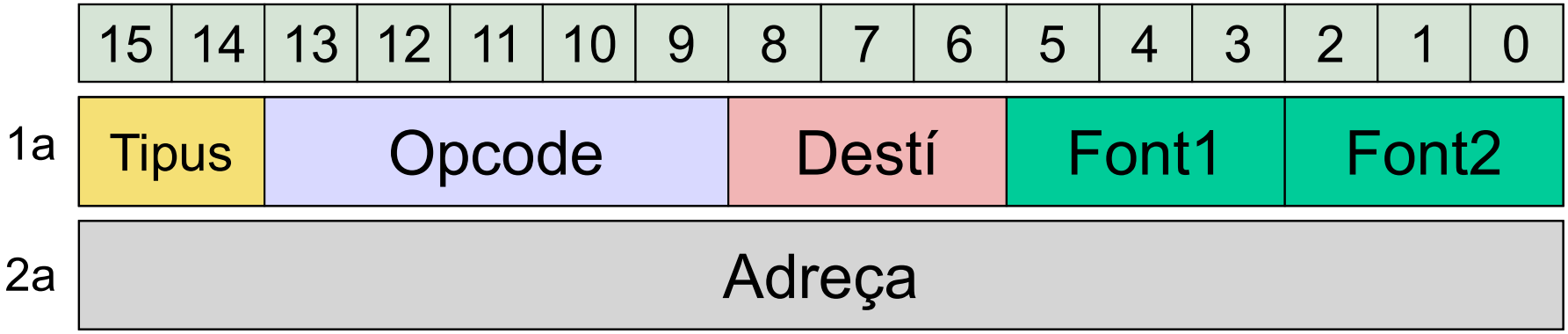
Instruccions de càrrega (*Load Instructions*)
Llegeix dada de posició de memòria i guarda en registre. Pex:

Load R2, A equival a : R2 <= Mem[A]

Instruccions d'emmagatzematge (*Store Instructions*)
Guarda contingut d'un registre a una posició de memòria.Pex:

Store A, R2 equival a : Mem[A] <= R2

INSTRUCCIONS DE MOVIMENT O DE MEMÒRIA (*Move Instructions*)



Load
Store

instrucció		acció
L imm Destí	Adreça	CR[Destí] <= Adreça
L dir Destí	Adreça	CR[Destí] <= Mem[Adreça]
S dir Font1	Adreça	Mem[Adreça] <= CR[Font1]
L rel Destí, Font2	Adreça	CR[Destí] <= Mem[CR[Font2] + Adreça]
S rel Font1, Font2	Adreça	Mem[CR[Font2] + Adreça] <= CR[Font1]
L in Destí	Adreça	CR[Destí] <= Mem[Mem[Adreça]]
S in Font1	Adreça	Mem[Mem[Adreça]] <= CR[Font1]

Instruccions de Memòria exemple

INSTRUCCIONS DE MOVIMENT O DE MEMÒRIA (*Move Instructions*)

LOAD Rd, Imm(Rf)

Desa el contingut que hi ha a la posició de memòria Imm + contingut de Rf en el registre Rd. Podem parlar de carregar un word **LW**, carregar 16 bits **LH** o carregar un byte **LB. També està la instrucció LA, que permet carregar una adreça en un registre**

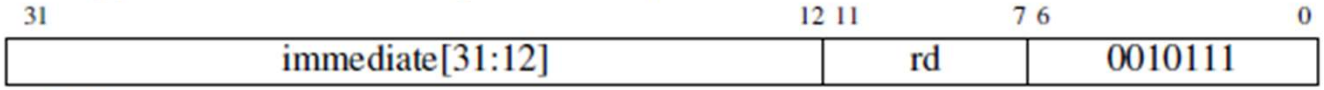
STORE Rf1, valor(Rf2)

Guarda el contingut de Rf1 en la posició de memòria Imm + contingut de Rf2. Igual que en el cas anterior les Instruccions poden treballar amb 8 bits **SB**, 16 bits **SH** o 32 bits **SW**

```
1  # Primer programa en assemblador
2  # Autor: Alumne
3
4  # memòria de dades
5  .data
6  xx: .word 3
7  yy: .word 5
8  res: .word 0
9
10 # memòria d'instruccions
11 .text
12 #permet guardar l'adreça associada a xx
13 #en el registre a0
14 la a0, xx
15 # observa com l'immediat augmenta de 4
16 # en 4...
17 lw a1, 0(a0)
18 lw a2, 4(a0)
19 add a3, a1, a2
20 sll a3, a3, a1
21 sw a3, 8(a0)
22 end: nop
23 j end
```

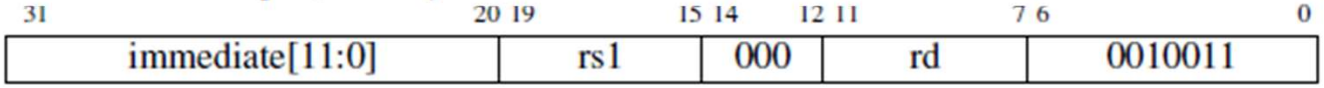
auipc rd, immediate `x[rd] = pc + sext(immediate[31:12] << 12)`

Add Upper Immediate to PC. Tipo U, RV32I y RV64I.

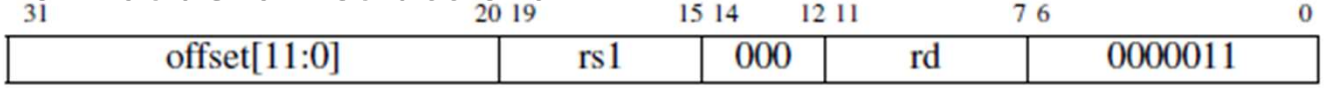


addi rd, rs1, immediate `x[rd] = x[rs1] + sext(immediate)`

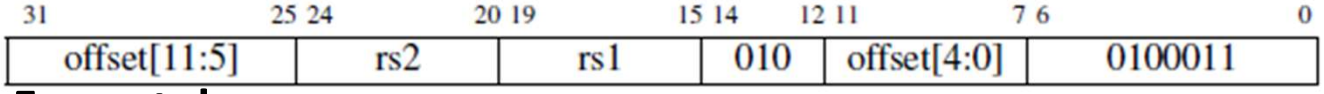
Add Immediate. Tipo I, RV32I y RV64I.



Format de la instrucció la



Format de lb



Format de sw

s

INSTRUCCIONS DE SALT CONDICIONAL I INCONDICIONAL (*Branch, Jump Instructions*)

Seleccionen la instrucció següent a ser executada en la UP

instruccions de salt Incondicional

El programa salta sempre a una adreça determinada

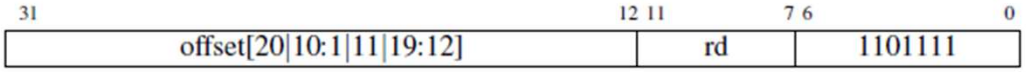
j offset `pc += sext(offset)`

Jump. Pseudoinstrucción, RV32I y RV64I.
Escribe al *pc* su valor actual más el *offset* extendido en signo. Se extiende a **jal** x0, offset.

---

jal rd, offset `x[rd] = pc+4; pc += sext(offset)`

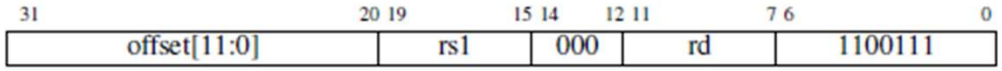
Jump and Link. Tipo J, RV32I y RV64I.
Escribe la dirección de la siguiente instrucción (*pc*+4) en x[*rd*], luego asigna al *pc* su valor actual más el *offset* extendido en signo. Si *rd* es omitido, se asume x1.
Formas comprimidas: **c.j** offset; **c.jal** offset



---

jalr rd, offset(rs1) `t=pc+4; pc=(x[rs1]+sext(offset))&~1; x[rd]=t`

Jump and Link Register. Tipo I, RV32I y RV64I.
Escribe x[*rs1*] + *sign-extend*(*offset*) al *pc*, enmascarando a cero el bit menos significativo de la dirección calculada, luego escribe el valor previo del *pc*+4 en x[*rd*]. Si *rd* es omitido, se asume x1.
Formas comprimidas: **c.jr** rs1; **c.jalr** rs1



---

jr rs1 `pc = x[rs1]`

Jump Register. Pseudoinstrucción, RV32I y RV64I.
Escribe x[*rs1*] al *pc*. Se extiende a **jalr** x0, 0(rs1).

INSTRUCCIONS DE SALT CONDICIONAL I INCONDICIONAL (*Branch, Jump Instructions*)

instruccions de salt Condicional

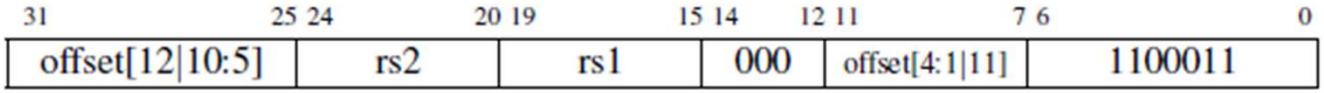
Si es compleix una determinada condició, fan saltar el programa a una instrucció, especificada en un operant. En cas contrari s'executa la instrucció següent de la seqüència.

beq rs1, rs2, offset `if (rs1 == rs2) pc += sext(offset)`

Branch if Equal. Tipo B, RV32I y RV64I.

Si el registro x[*rs1*] es igual al registro x[*rs2*], asignar al *pc* su valor actual más el *offset* sign-extended.

Forma comprimida: **c.beqz** rs1,offset



beqz rs1, offset `if (rs1 == 0) pc += sext(offset)`

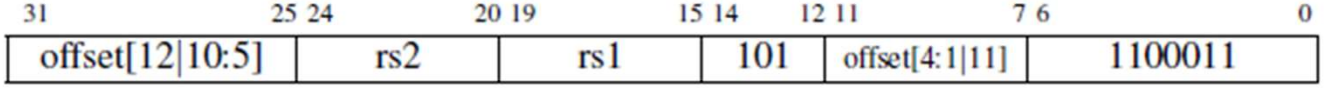
Branch if Equal to Zero. Pseudoinstrucción, RV32I y RV64I.

Se extiende a **beq** rs1, x0, offset.

bge rs1, rs2, offset `if (rs1 ≥s rs2) pc += sext(offset)`

Branch if Greater Than or Equal. Tipo B, RV32I y RV64I.

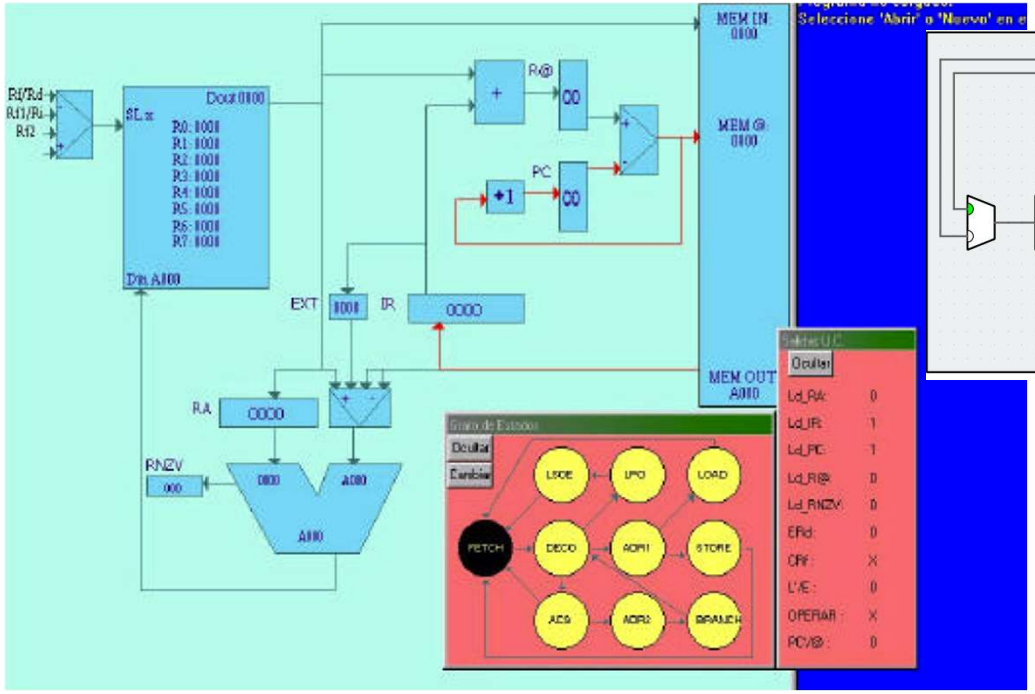
Si el registro x[*rs1*] es por lo menos x[*rs2*], tratando los valores como números de complemento a dos, asignar al *pc* su valor actual más el *offset* sign-extended.



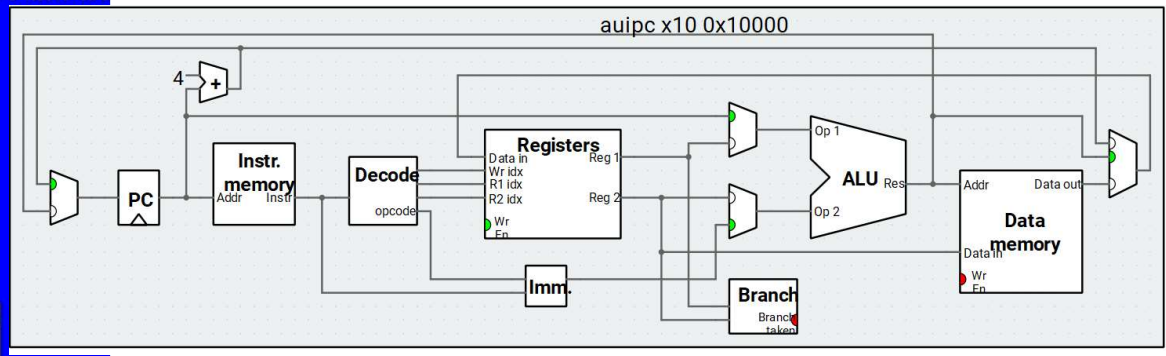
INSTRUCCIONS DE SALT CONDICIONAL I INCONDICIONAL (*Branch, Jump Instructions*)

Dues formes usuals de **comprovar la condició** d'una instrucció de **salt condicional**.

1. Registre d'estat
2. Comparació de valors guardats a registres



CPU amb registre d'estat



CPU sense registre d'estat explícit

INSTRUCCIONS DE SALT CONDICIONAL I INCONDICIONAL (*Branch, Jump*)

1.Codi de condició que s'actualitza quan s'executen algunes operacions; generalment estan continguts en un **registre d'estat**.

Pex, resultat d'una operació aritmètica (*Add, Substract. ..*) Pot fixar un codi de condició amb quatre possibles valors,
zero, positiu,
negatiu, desbordament.

Així podem tenir 4 instruccions de salt condicional:

Brp A Saltar a posició A si resultat positiu.
Brn A Saltar a posició A si resultat negatiu.
Brz A Saltar a posició A si resultat zero.
Bro A Saltar a posició A si produït desbordament

resultat de darrera operació exe. que afecti el codi d'estatus.

INSTRUCCIONS DE SALT CONDICIONAL I INCONDICIONAL (*Branch, Jump*)

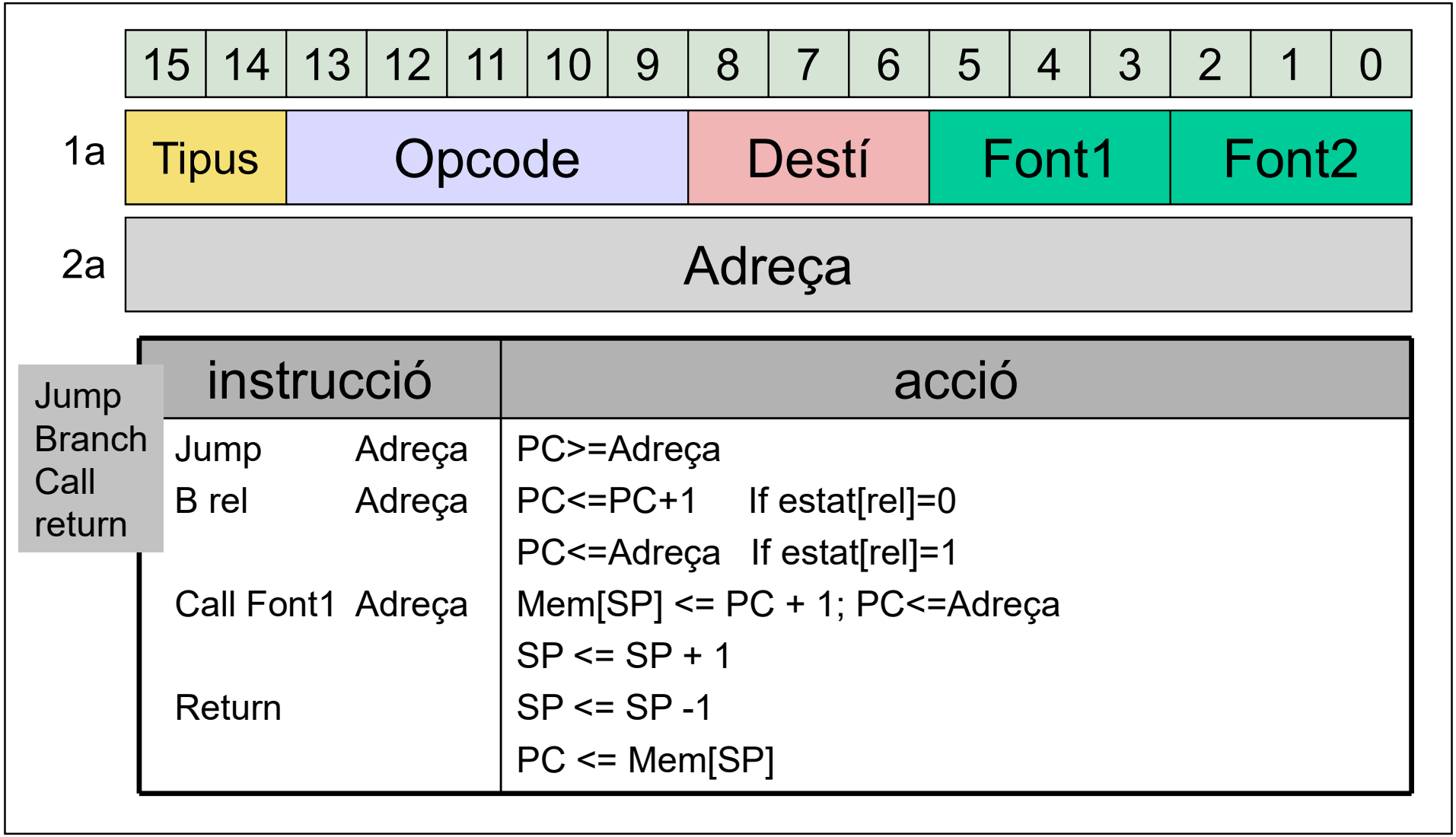
1. Un format d'instrucció de 3 camps.
   Compara els valors de dos registres i decideix la següent instrucció en base a la comparació;
   La comparació es fa a la mateixa instrucció.

Per exemple, la instrucció **Salta-quan-igual** (*Branch-on-equal o Beq*) comprova si els dos dels registres (R2 i R3) són iguals i si ho són executa la instrucció ubicada a la posició de memòria especificada (A):

Beq R2, R3, A

compara [R2] i [R3] i si són =
executa instrucció situada a pos. de memòria *Mem*[A]

INSTRUCCIONS DE SALT CONDICIONAL I INCONDICIONAL (*Branch, Jump*)



Jump
Branch
Call
return

instrucció		acció
Jump	Adreça	PC>=Adreça
B rel	Adreça	PC<=PC+1 If estat[rel]=0
		PC<=Adreça If estat[rel]=1
Call Font1	Adreça	Mem[SP] <= PC + 1; PC<=Adreça
		SP <= SP + 1
Return		SP <= SP -1
		PC <= Mem[SP]

Instruccions de salt exemple

Algunes Instruccions de Ripes

AL

ADD a1,a2,a3

SUB a1,a2,a3

ADDI a1, a2,a3

XOR a1, a2, a3

SLL a3, a1, a2 SRL a1,a2,a3

AND a1, a2, a3

@MEM

LW a1, VALUE(a2)

SW a1, value(a2)

@BRANCH

BR @MEM

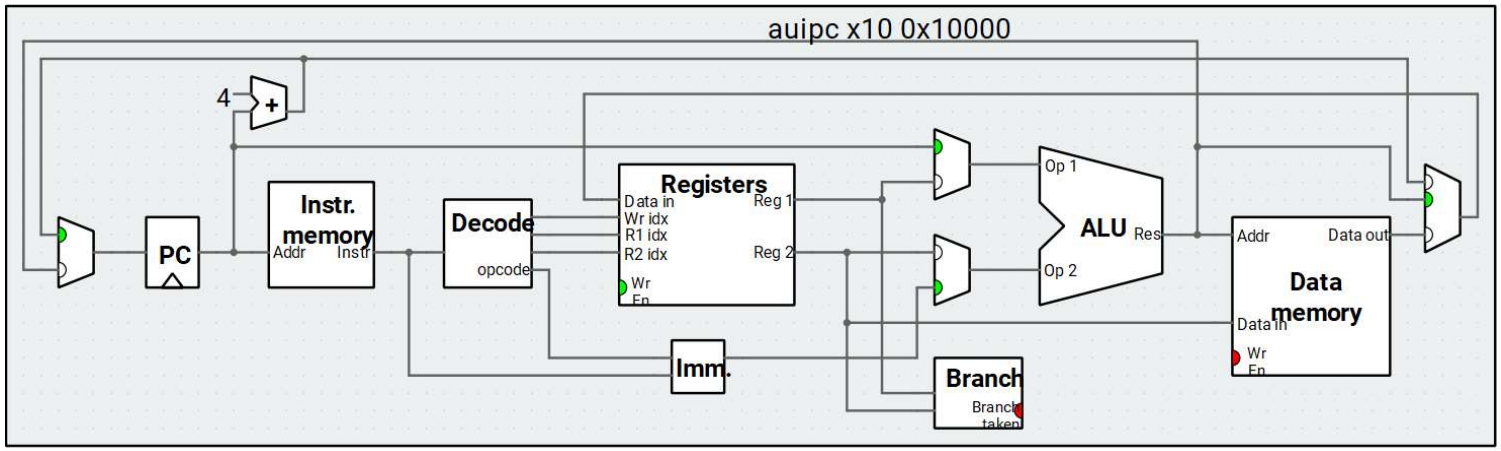
BEQ a1, a2, IMM

BNE a1, a2, IMM

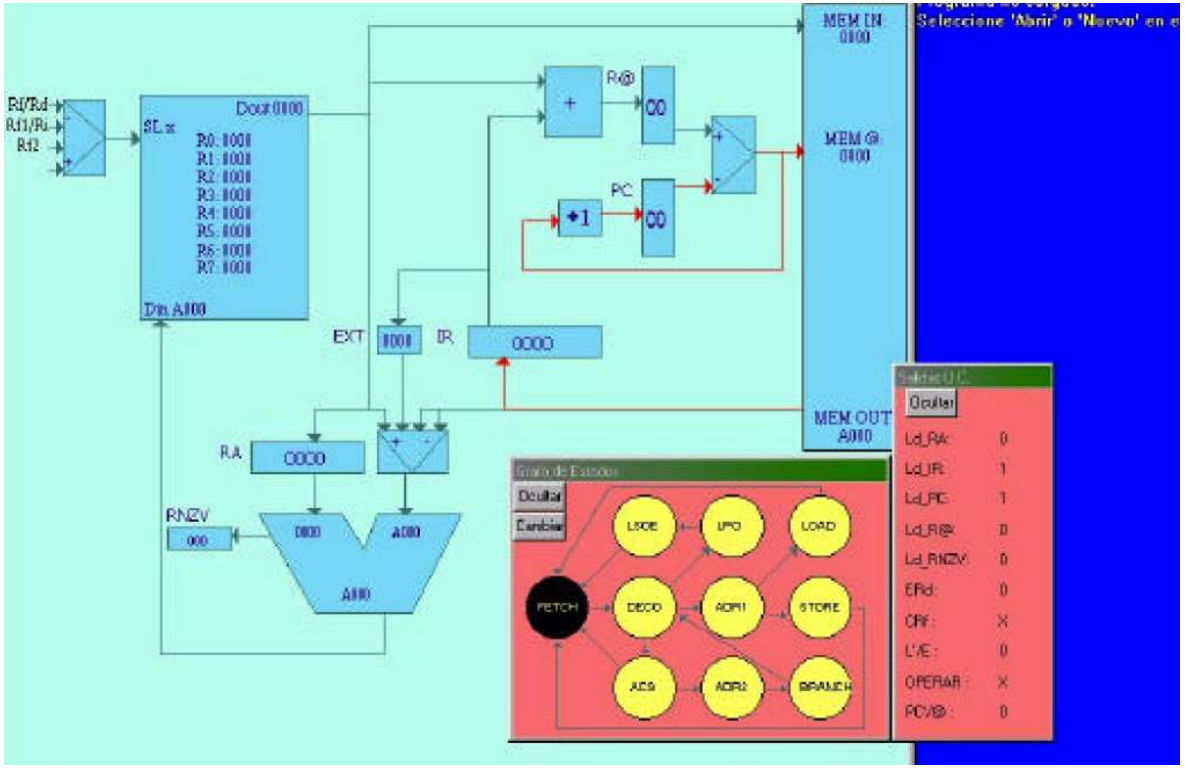
BLT a1, a2, IMM

J IMM

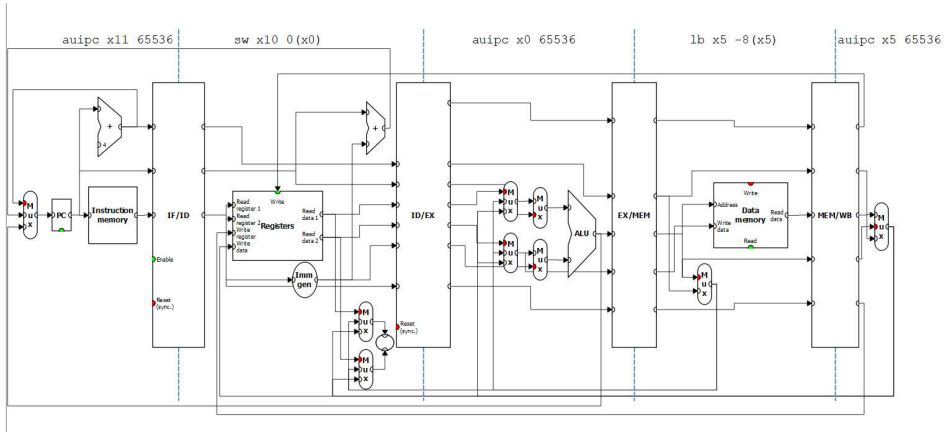
BGE a1, a2, @MEM



Estructura Von Neumann. Arquitectura ISA - SiMR



Estructura Harvard. ISA Ripes_RISC-V



CONJUNT D'INSTRUCCIONS

En definir un conjunt d'instruccions és important considerar quants camps d'adreça podran contenir les instruccions.

El # de camps d'adreça afecta la longitud del programa i les prestacions del processador.

+camps d'adreça ➔ + llargada ➔ - programa (- instruccions)
però
instruccions + llargues ➔ +accessos a memòria per portar instruccions i operands.

Nombre d'accessos a memòria: paràmetre per avaluar les prestacions.

Les memòries són + lentes que el processador, per tant és un paràmetre per avaluar la velocitat d'execució d'un programa

CONJUNT D'INSTRUCCIONS

•Es dissenya el **CONJUNT D'INSTRUCCIONS** de manera que el nombre d'accessos a memòria sigui mínim.

•Instruccions amb diferent nombre de camps, des **d'1 a 3**

•Les instruccions de 3 camps són (en principi) les + potents, contenen la posició dels operands i del resultat.

•En general no més de 3 camps, evitant que aquests siguin de posició en memòria. Els recursos de l'UP són de 2 entrades.

•Per altra banda una instrucció de 3 adreces de memòria ocupa molt d'espai, com sempre **cal un compromís**.

Nombre d'adreces i accessos a memòria

Exemple:

•Analitzarem instruccions de diferent tipus i nombre d'adreces per computar l'expressió:

$$c = (a+b)*(a-b)$$

•Variables a,b,c i variable temporal x en posicions de memòria A, B, C i X, respectivament.

•Espai memòria: 16MB : 24 b adreces

•BUS 32 bits

El num. de bits que requereix una instrucció depén de la capacitat total de memòria

Instruccions de 3 adreces

3 adreces	Opcode	Total	Num. Acc/instr
72b	8 b	80 b	3*32 = 96>80

3x24

Mem: 24 b
BUS: 32b

Total: **3** accessos/instrucció

Accessos a memòria

programa	Fetch	Exe
1. *Add* X, A, B (Mem[X] <= Mem[A] + Mem[B])	3 acces.	3 acc.
2. *Sub* C, A, B (Mem[C] <= Mem[A] - Mem[B])	3 acces	3 acc.
3. *Mul* C, X, C (Mem[C] <= Mem[X] * Mem[C])	3 acces	3 acc.

Total 18 accessos a memòria

Instruccions de 2 adreces

el 1r op. i resultat comparteixen mateixa adreça de memòria.

2 adreces	Opcode	Total	Num. Acc/instr
48b	8 b	56 b	2*32 = 64

Mem: 24 b
BUS: 32b

2x24 Total: **2** accessos/instrucció

Programa		Fetch	Exe
Move X,A	(Mem[X] <= Mem[A])	2 acc	2 acc (=>A , X<=)
Add X,B	(Mem[X] <= Mem[X] + Mem[B])	2 acc	3 acc (B,X ; X)
Move C,A	(Mem[C] <= Mem[A])	2 acc	2 acc
Sub C, B	(Mem[C] <= Mem[C] - Mem[B])	2 acc	3 acc (C,B ; C)
Mul C, X	(Mem[C] <= Mem[C] * Mem[X])	2 acc	3 acc (C,X ; C)

10 acc. per portar instruccions i
13 per llegir/guardar operands i resultats, **Total 23 accessos a memòria**

Per tant, temps d'execució és major.
Hem necessitat 2 instr. extra de tipus *Move* (la 1 i la 3)

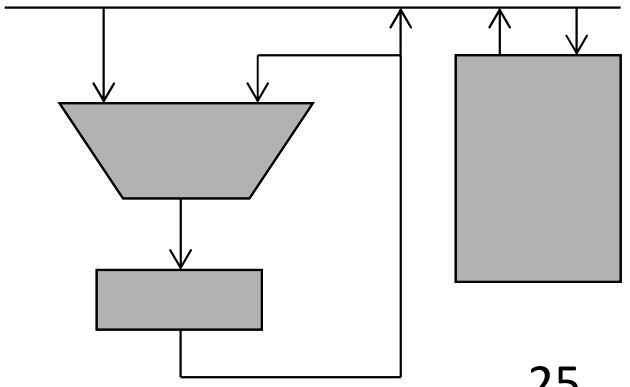
Instruccions de 1 adreça i acumulador (ACC) dedicat

Reducció de nombre d'accessos. ACC guarda resultat d'operació i algun operand

Programa : 7 instruccions **Accessos a memòria**

Programa	Fetch	Exe
Load A (ACC <= Mem[A])	1 acc	1 acc
Add B (ACC <= ACC + Mem[B])	1 acc	1 acc
Store X (Mem[X] <= ACC)	1 acc	1 acc
Load A (ACC <= Mem[A])	1 acc	1 acc
Sub B (ACC <= ACC - Mem[B])	1 acc	1 acc
Mul X (ACC <= ACC * Mem[X])	1 acc	1 acc
Store C (Mem[C] <= ACC)	1 acc	1 acc

Total 14 accessos a memòria



Instruccions de 2 adreces amb Conjunt de Registres (CR)

•Caldrà incloure camps d'adreces addicionals per adreçar el CR.

•Considerem el programa amb instruccions de dues adreces, on, una de les adreces especifica una posició del (CR),

•El Conjunt de Registres per aquest programa exemple disposa d'almenys cinc registres

Instruccions de 2 adreces amb Conjunt de Registres

Programa : 6 instruccions

Accessos a memòria

Programa		Fetch	Exe
Load R1,A	(CR[1] <= Mem[A])	2 acc	1 acc
Load R2,B	(CR[2] <= Mem[B])	2 acc	1 acc
Add R3, R1, R2	(CR[3] <= CR[1] + CR[2])	1 acc	
Sub R4, R1, R2	(CR[4] <= CR[1] - CR[2])	1 acc	
Mul R5, R3, R4	(CR[5] <= CR[3] * CR[4])	1 acc	
Store C, R5	(Mem[C] <= CR[5])	2 acc	1 acc

Total 12 accessos a memòria

Instruccions de 2 adreces amb Conjunt de Registres

•Només les Instructions de *Load* i *Store* accedeixen a memòria,
•Instruccions aritmètiques només accedeixen a regs ➔ camp adreça + curt

El CR petit : accés amb pocs bits d'adreça.

Cal usar
• instruccions de 3 camps on els operands estiguin en regs i
• instruccions de 2 adreces en les d'accés a memòria.

Aquesta estratègia usada per la majoria de processadors:
Instruccions més curtes i menor freqüència d'accessos

Estratègia avalada pel fet que cada variable en el programa s'usa més d'un cop, i llegir-les des d'un CR intern i ràpid redueix el nombre d'accessos a la memòria principal + lenta

Resum d'accessos/instruccions

instruccions	Longitud ins.	# instr	Total bits	# acc. mem
In-3adr	80 b	3	288	18
In-2adr	56 b	5	320	23
In-1adr+ACC	32 b	7	224	14
In-1adr-CR	32 b	6	216	**12**

MODES D'ADREÇAMENT DE MEMÒRIA

camp d'adreça: conté informació necessària per:

- determinar posició d'operands
- determinar posició del resultat de l'operació.

Diversos modes d'interpretar la informació de camp d'adreça.

•Diferents modes d'adreçament,
•Reduir el camp d'adreça, especificant només part de l'adreça, mentre el "mode" defineix com calcular l'adreça sencera.

•En general, es necessiten modes d'adreçament per suportar diferents construccions de llenguatges de programació, estructures de dades i tasques de SOs: bucles, punters de dades, reubicació de programa i commutació de contextos.

MODES D'ADREÇAMENT DE MEMÒRIA

La Disponibilitat en modes d'adreçament proporciona al rogramador la possibilitat d'escriure programes eficients en nombre d'instruccions i temps d'execució.

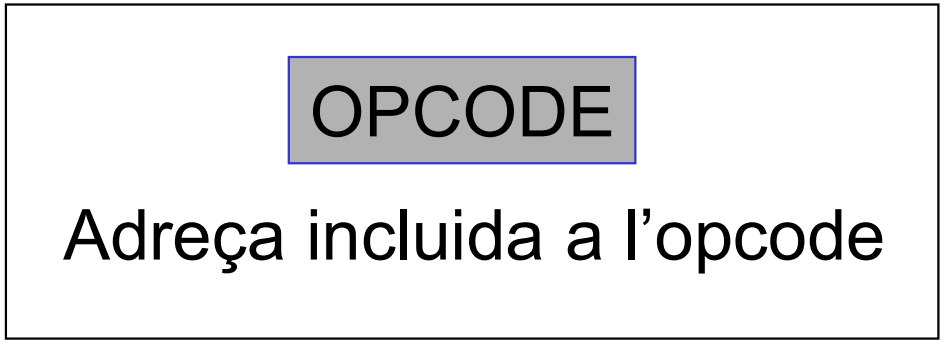
Els modes d'adreçament més comuns són:

1. Mode d'adreçament implícit
2. Mode d'adreçament immediat
3. Mode d'adreçament directe
4. Mode d'adreçament indirecte
5. Mode d'adreçament relatiu
6. Mode d'adreçament Indexat

1. Mode d'adreçament implícit

•No necessita camp explícit d'adreça pq la ubicació de l'operand o el resultat ja està especificada a l'opcode.

•Pex. instruccions de clear del registre d'estatus o l'acumulador. registres són únics i implícits en l'opcode (aquests registres solen ser únics i estan per tant implícits en l'opcode)

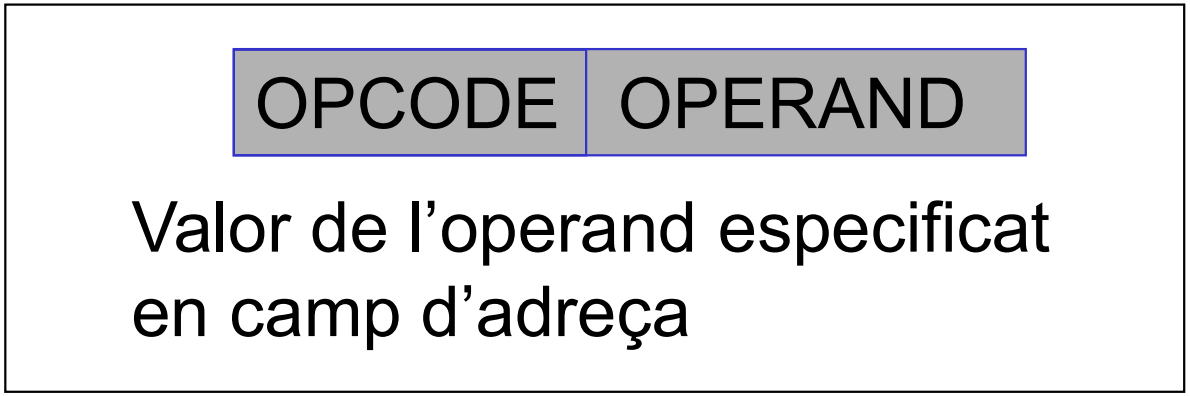


Adreçament Implícit

2. Mode d'adreçament immediat

•Operand especificat en camp d'adreça.
•Una instrucció d'adreçament immediat té un camp d'operand en el lloc del camp d'adreça

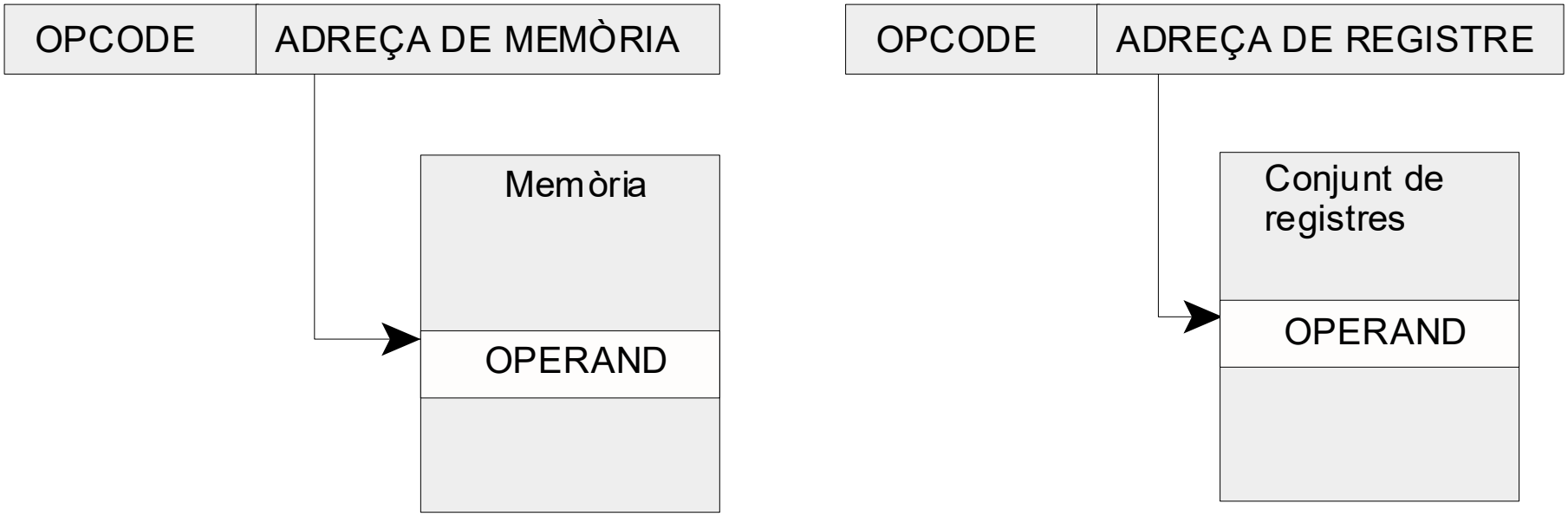
•Per especificar constants usades com operands en l'operació. Quan es treballa amb **constants** que poden ser subministrades en camp d'adreça en lloc d'accedir-les des de la memòria, estalvia, per tant, accessos a memòria.



Adreçament IMMEDIAT

3. Mode d'adreçament directe

- El Camp Adreça especifica la posició de l'operand
- Adreçament directe a memòria o a registre del CR
- Cal tenir en compte que l'adreça de memòria sempre serà molt més gran que la del CR



Adreçament DIRECTE

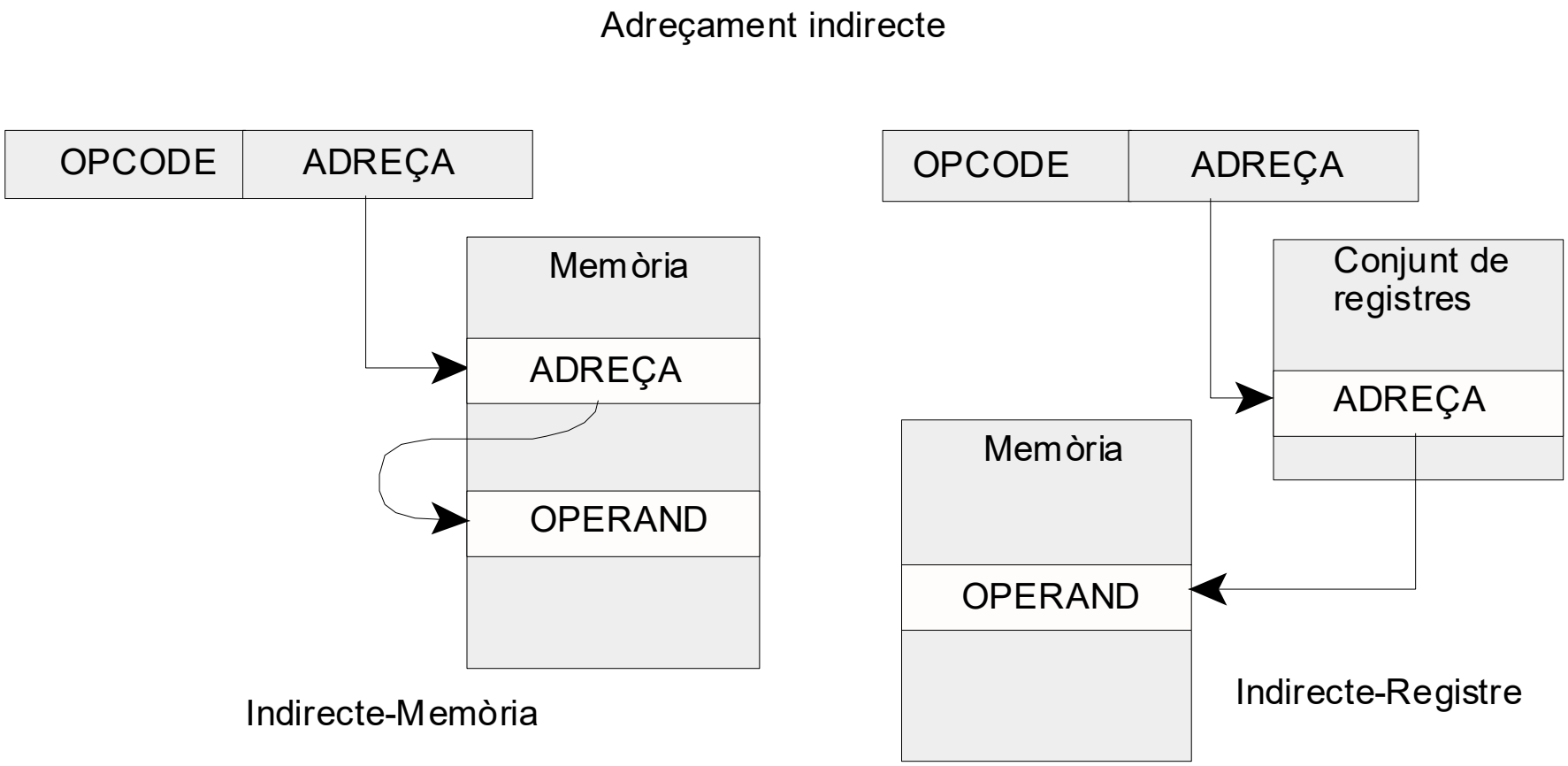
4. Mode d'adreçament indirecte

•Camp adreça especifica la ubicació de l'adreça de l'operand

•El processador ha d'accedir a memòria el doble de cops que en l'adreçament directe, 1 per llegir l'adreça i 1 per llegir l'operand o guardar el resultat

•En el mode **indirecte-registre** el camp d'adreça conté la direcció del registre que conté l'adreça de l'operand; el programador s'ha d'assegurar que l'adreça està en el registre adequat, abans d'accedir-hi

4. Mode d'adreçament indirecte

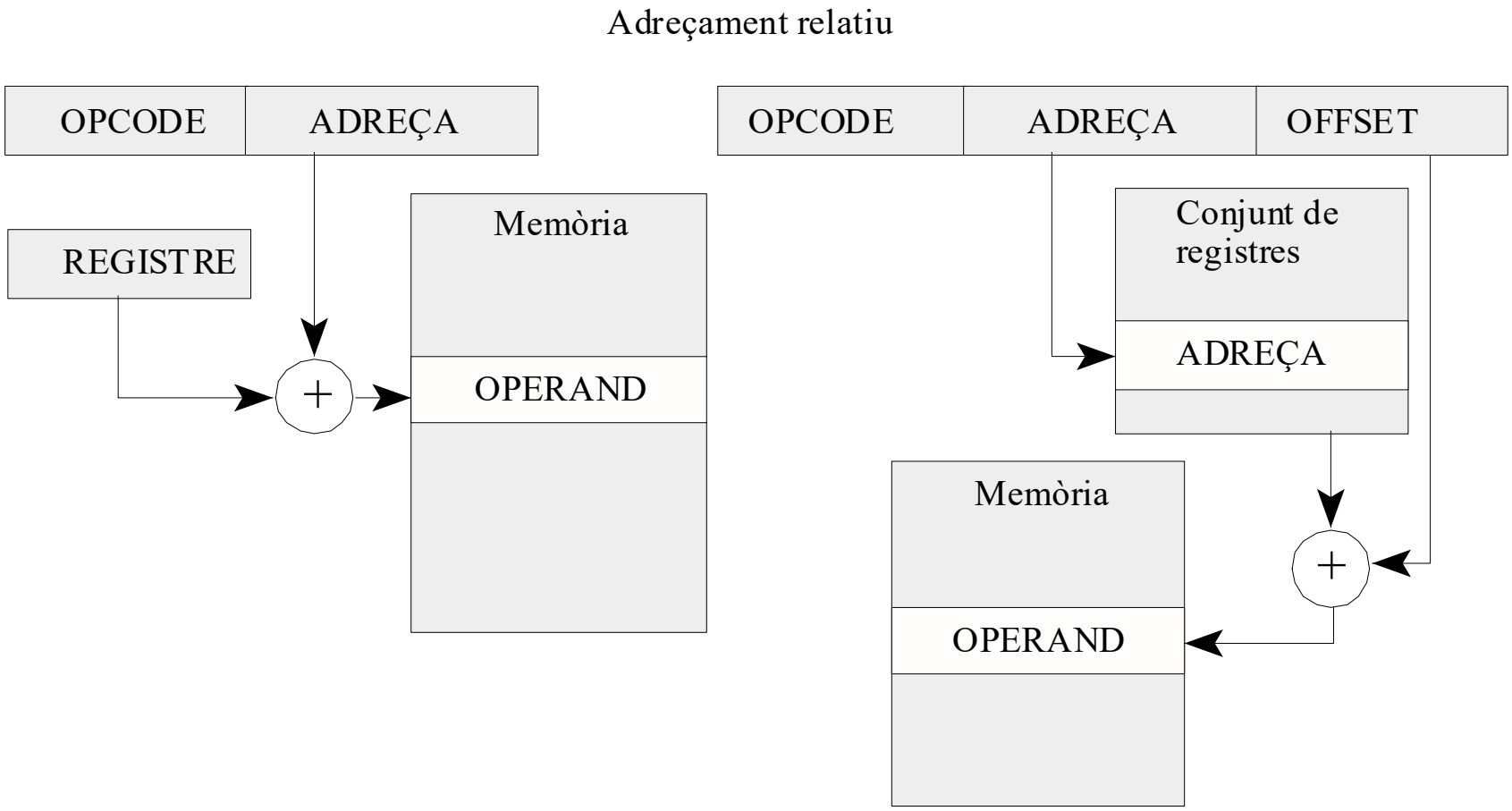


Adreçament indirecte

5. Mode d'adreçament relatiu

•Camp adreça=*offset*, és sumat al contingut d'un registre especificat, com el (PC) o un registre del (CR).
•L'*offset* és enter (positiu o negatiu). Quan l'*offset* es suma al PC la suma és l'adreça d'una instrucció de posició propera a la instrucció apuntada pel PC.
•Per tant aquest mode és usat en instruccions de salt condicional, ja que l'adreça de salt condicional està generalment a prop de la pròpia instrucció de salt.
•També l'adreçament relatiu es pot utilitzar respecte a qualsevol registre del CR, en aquest cas implementa taules *Look-up* en què el registre conté l'origen de la taula i l'*offset* apunta a un element específic

5. Mode d'adreçament relatiu



Adreçament RELATIU

6. Mode d'adreçament Indexat

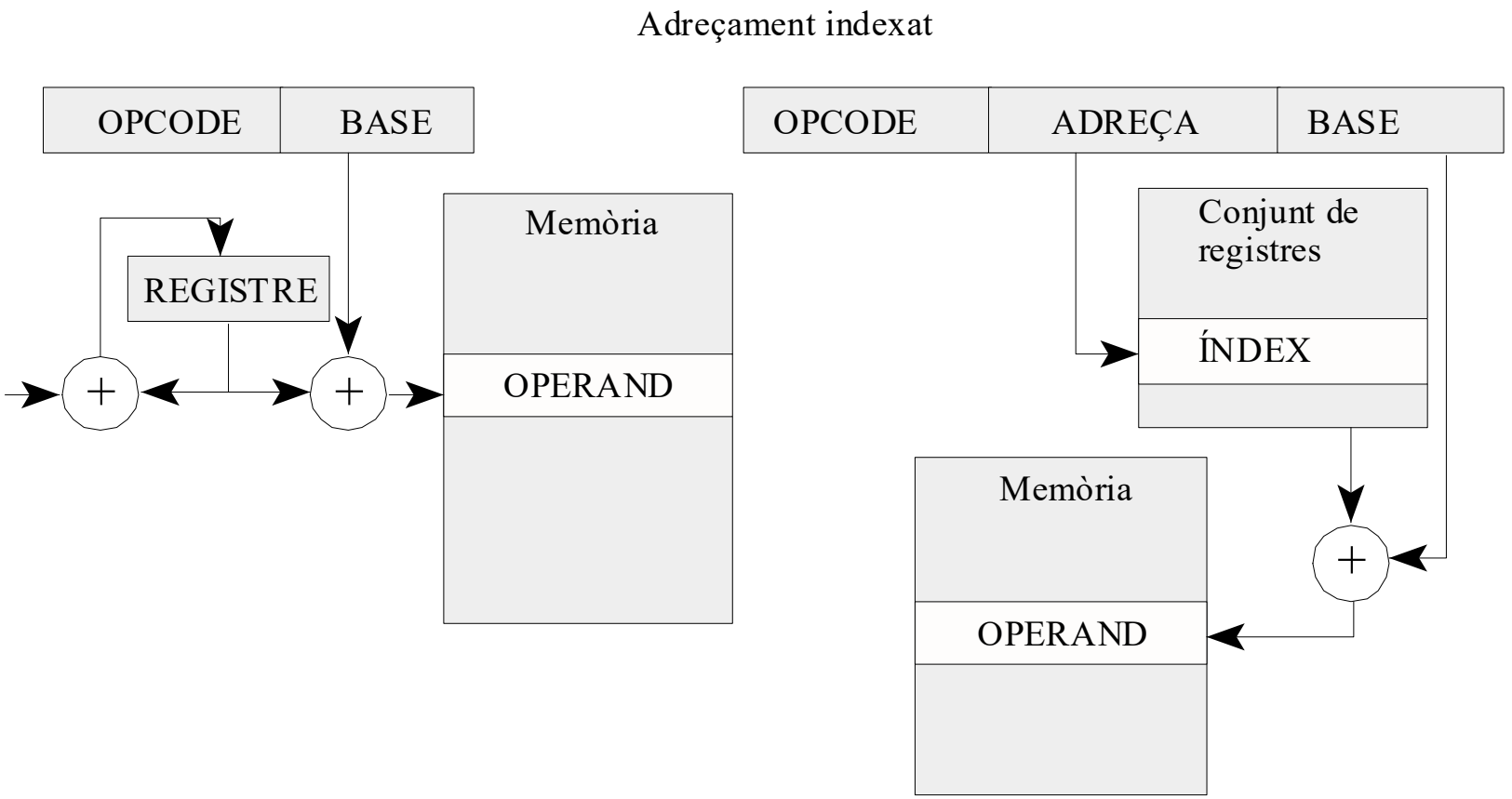
•L'adreçament indexat s'usa quan es necessita accedir a dades guardades en vectors (*arrays*), piles o cues.
•L'adreça especifica una adreça inicial, anomenada base, l'índex d'una dada particular s'especifica en un registre dedicat d'índex o en un del CR.
•Càlcul d'adreça efectiva: R(base) + R(índex)
•En algunes instruccions el valor del R(índex) és incrementat o decrementat automàticament per accedir al proper element del vector.
•Aquest tipus d'instrucció s'anomena instrucció autoincrement o autodecrement. També contribueix a reduir el nombre de bits necessaris en el camp adreça.

6. Mode d'adreçament Indexat

•El mode indexat és similar al relatiu, difereixen només en les posicions de la base i de l'índex o l'*offset*.

•En relatiu la base està en un registre dedicat i l'*offset* està en el camp adreça;
•En indexat la base està en el camp adreça i l'índex està en un registre.
•El mode relatiu es sol utilitzar en salts i l'indexat en accés a dades.

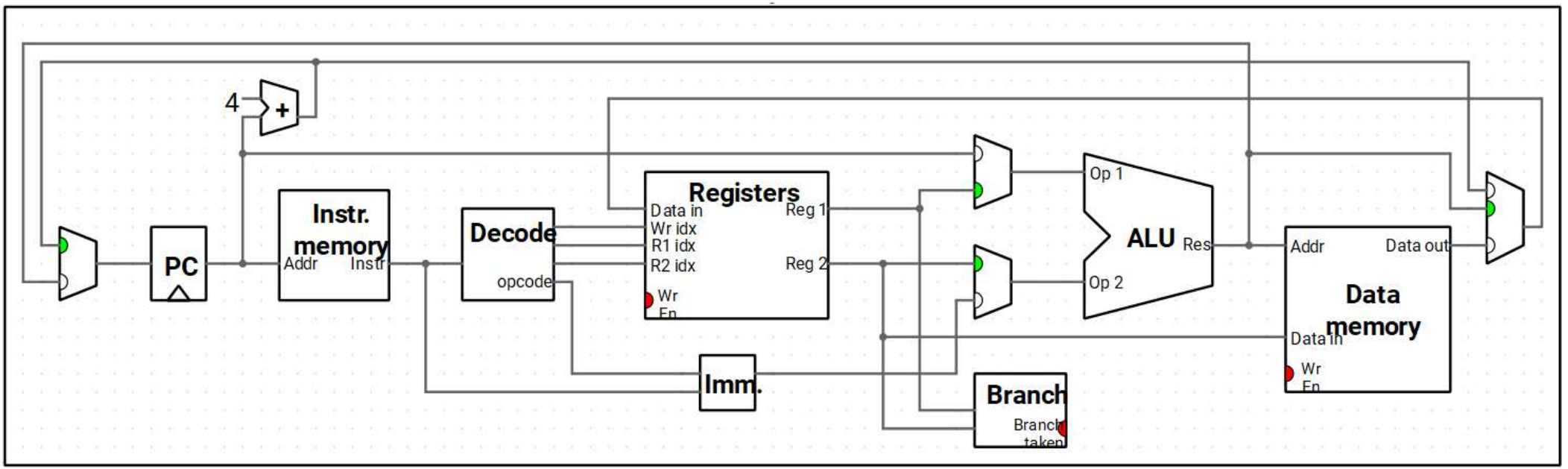
6. Mode d'adreçament Indexat



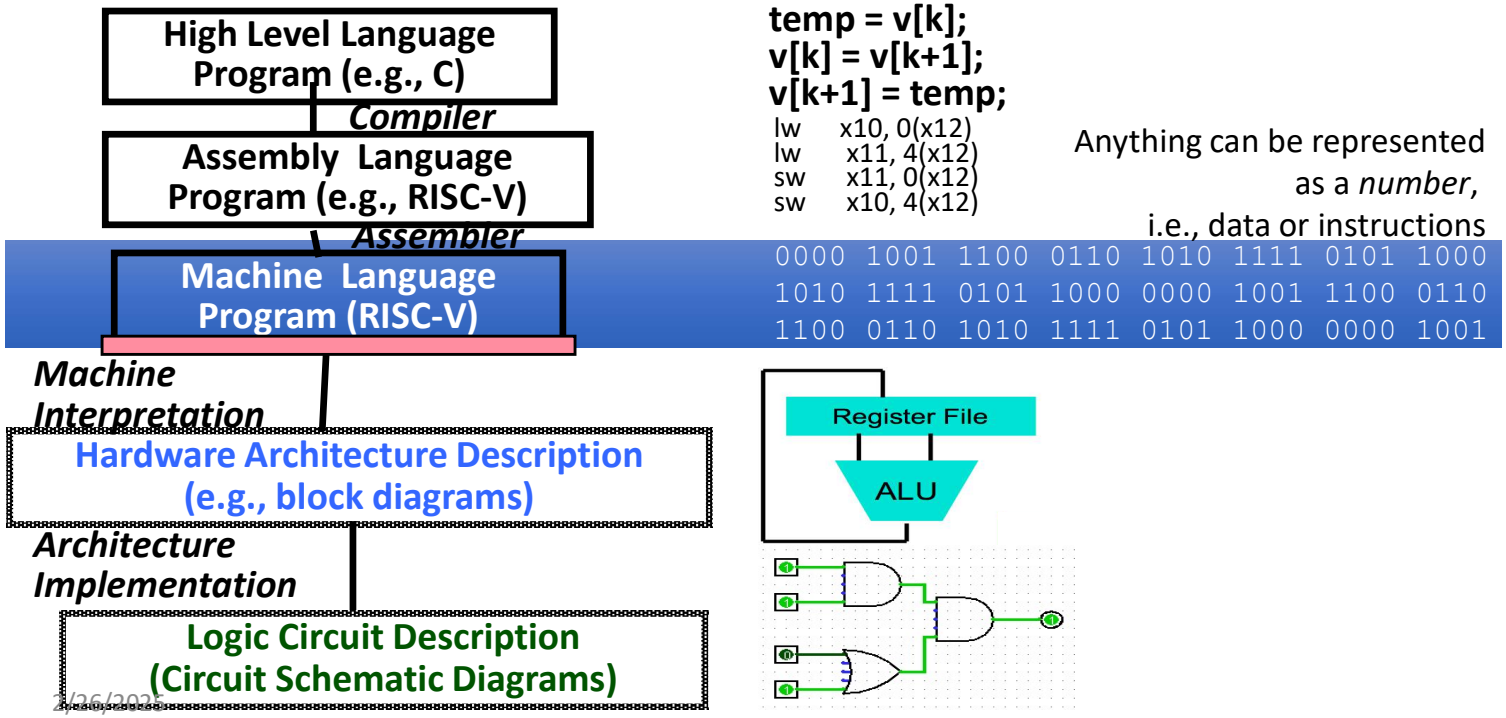
MODES D'ADREÇAMENT DE MEMÒRIA

En general la disponibilitat de diversos modes d'adreçament en un conjunt d'instruccions fa l'execució de programes més ràpida, però també incrementa la complexitat del processador.

El format d'instruccions des del punt de vista de RISC-V



Nivells

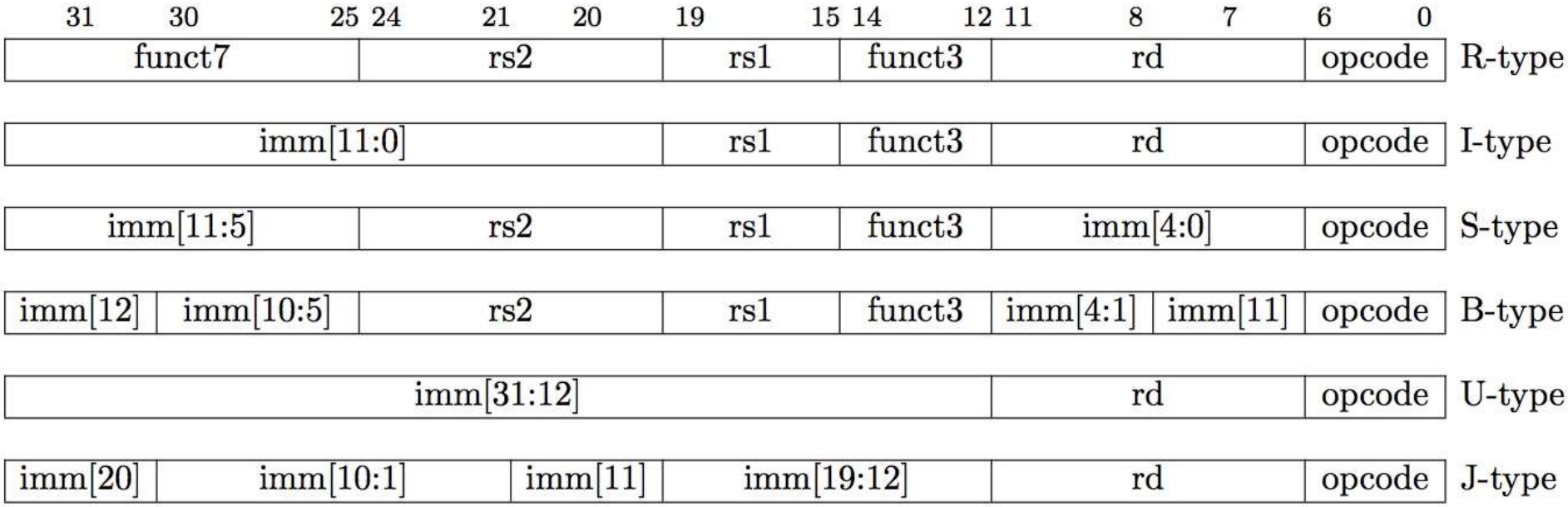


Anything can be represented as a *number*, i.e., data or instructions

Format de les instruccions

formats:

- R-format for register-register arithmetic operations
- I-format for register-immediate arithmetic operations and loads
- S-format for stores
- B-format for branches (minor variant of S-format, called SB before)
- U-format for 20-bit upper immediate instructions
- J-format for jumps (minor variant of U-format, called UJ before)


31 30 25 24 21 20 19 15 14 12 11 8 7 6 0
funct7 rs2 rs1 funct3 rd opcode R-type
imm[11:0] rs1 funct3 rd opcode I-type
imm[11:5] rs2 rs1 funct3 imm[4:0] opcode S-type
imm[12] imm[10:5] rs2 rs1 funct3 imm[4:1] imm[11] opcode B-type
imm[31:12] rd opcode U-type
imm[20] imm[10:1] imm[11] imm[19:12] rd opcode J-type

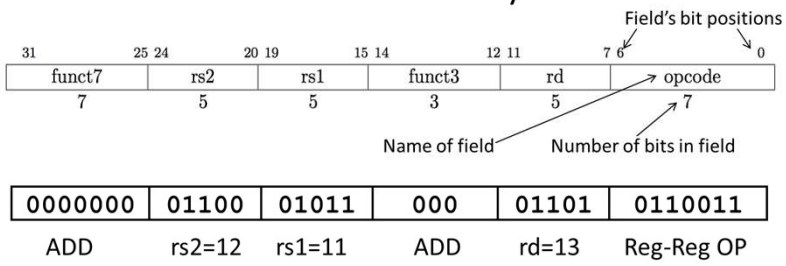
Tipus d'instruccions

Operacions amb registres:

- Format: Operació Registre destí, Registre font1, Registre font 2

Exemple:

ADD a3, a2, a1



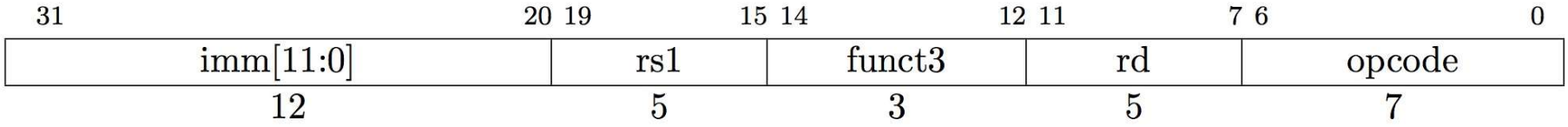
Register	ABI	Use by convention	Preserved?
x0	zero	hardwired to 0, ignores writes	*n/a*
x1	ra	return address for jumps	no
x2	sp	stack pointer	yes
x3	gp	global pointer	*n/a*
x4	tp	thread pointer	*n/a*
x5	t0	temporary register 0	no
x6	t1	temporary register 1	no
x7	t2	temporary register 2	no
x8	s0 *or* fp	saved register 0 *or* frame pointer	yes
x9	s1	saved register 1	yes
x10	a0	return value *or* function argument 0	no
x11	a1	return value *or* function argument 1	no
x12	a2	function argument 2	no
x13	a3	function argument 3	no
x14	a4	function argument 4	no

Register	ABI	Use by convention	Preserved?
x15	a5	function argument 5	no
x16	a6	function argument 6	no
x17	a7	function argument 7	no
x18	s2	saved register 2	yes
x19	s3	saved register 3	yes
x20	s4	saved register 4	yes
x21	s5	saved register 5	yes
x22	s6	saved register 6	yes
x23	s7	saved register 6	yes
x24	s8	saved register 8	yes
x25	s9	saved register 9	yes
x26	s10	saved register 10	yes
x27	s11	saved register 11	yes
x28	t3	temporary register 3	no
x29	t4	temporary register 4	no
x30	t5	temporary register 5	no
x31	t6	temporary register 6	no
pc	*(none)*	program counter	*n/a*

Operacions amb registres

0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

Instruccions amb Immediats



- Only one field is different from R-format, rs2 and funct7 replaced by 12-bit signed immediate, **`imm[11:0]`**
- Remaining fields (rs1, funct3, rd, opcode) same as before
- imm[11:0] can hold values in range [-2048_{ten} , $+2047_{ten}$]
- Immediate is always sign-extended to 32-bits before use in an arithmetic operation
- We'll later see how to handle immediates > 12 bits

Instruccions amb Immediats

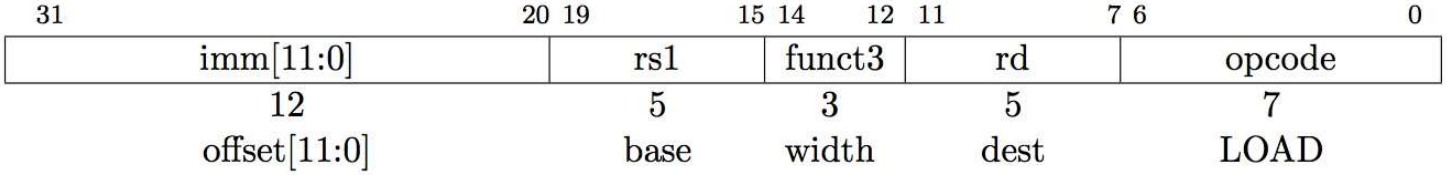
All RV32 I-format Arithmetic Instructions

imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI

One of the higher-order immediate bits is used to distinguish “shift right logical” (SRLI) from “shift right arithmetic” (SRAI)

“Shift-by-immediate” instructions only use lower 5 bits of the immediate value for shift amount (can only shift by 0-31 bit positions)

Load Instructions are also I-Type



The 12-bit signed immediate is added to the base address in register rs1 to form the memory address

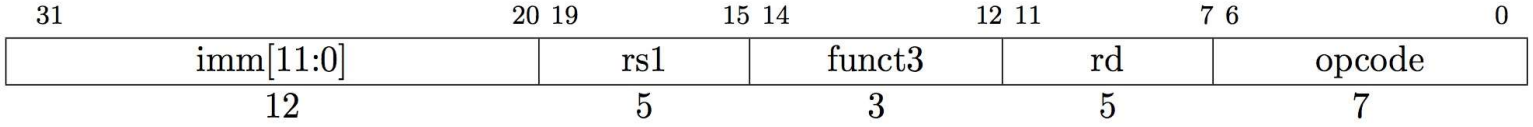
- This is very similar to the add-immediate operation but used to create address not to create final result

The value loaded from memory is stored in register rd

I-Format Load Example

RISC-V Assembly Instruction:

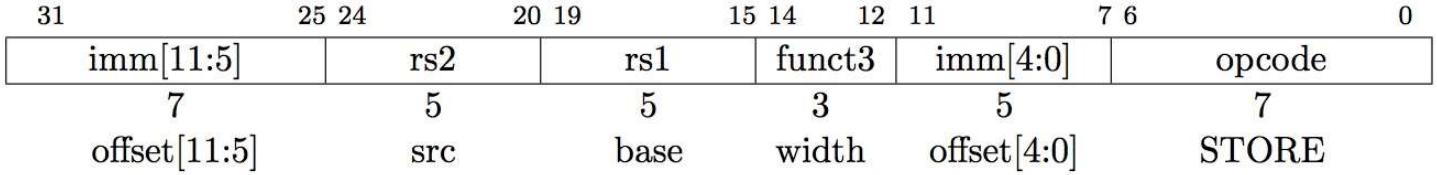
```
lw x14, 8(x2)
```



000000001000	00010	010	01110	0000011
imm=+8	rs1=2	LW	rd=14	LOAD

imm[11:0]	rs1	000	rd	0000011	LB
imm[11:0]	rs1	001	rd	0000011	LH
imm[11:0]	rs1	010	rd	0000011	LW
imm[11:0]	rs1	100	rd	0000011	LBU
imm[11:0]	rs1	101	rd	0000011	LHU

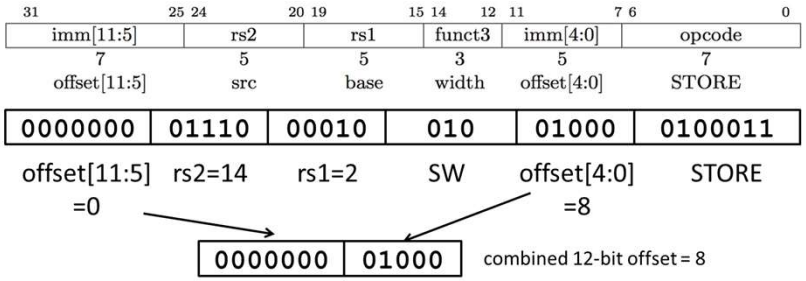
S-Format Used for Stores



- Store needs to read two registers, rs1 for base memory address, and rs2 for data to be stored, as well as need immediate offset!
- Can't have both rs2 and immediate in same place as other instructions!
- Note that stores don't write a value to the register file, ***no rd***!
- RISC-V design decision is move low 5 bits of immediate to where rd field was in other instructions – keep rs1/rs2 fields in same place
  - register names more critical than immediate bits in hardware design

- RISC-V Assembly Instruction:

```
sw x14, 8(x2)
```



imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW

RISC-V Conditional Branches

E.g., **`BEQ x1, x2, Label`**

Branches read two registers but don't write a register (similar to stores)

How to encode label, i.e., where to branch to?

- PC-Relative Addressing: Use the `immediate` field as a two's-complement offset to PC
  - Branches generally change the PC by a small amount
  - Can specify ± 2^{11} addresses from the PC
- Why not use byte address offset from PC?

RISC-V Conditional Branches

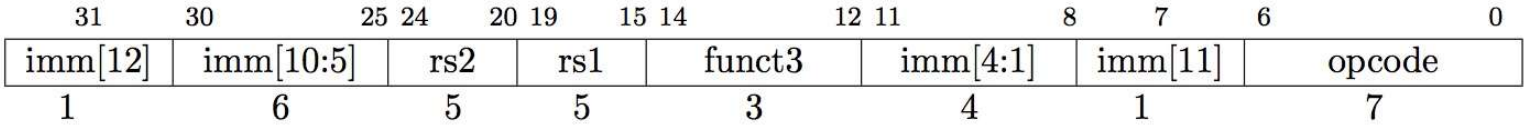
One idea: To improve the reach of a single branch instruction, multiply the offset by four bytes before adding to PC

This would allow one branch instruction to reach $\pm 2^{11} \times$ 32-bit instructions either side of PC

- Four times greater reach than using byte offset

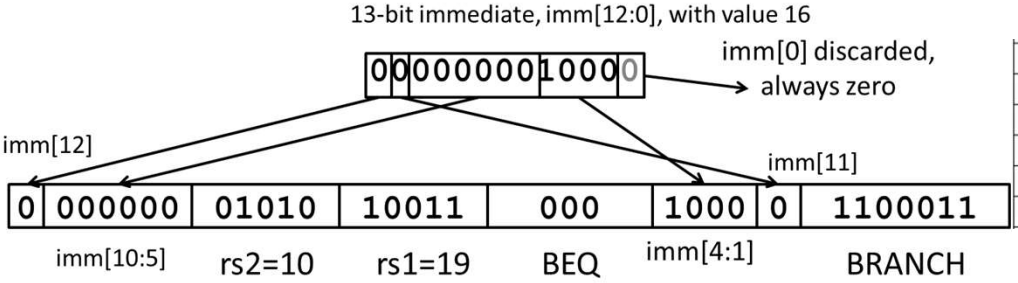
- If we don't take the branch:
  `PC = PC + 4` (i.e., next instruction)
- If we do take the branch:
  `PC = PC + immediate*4`
- **Observations:**
  - `immediate` is number of instructions to jump (remember, specifies words) either forward (+) or backwards (–)

RISC-V Conditional Branches



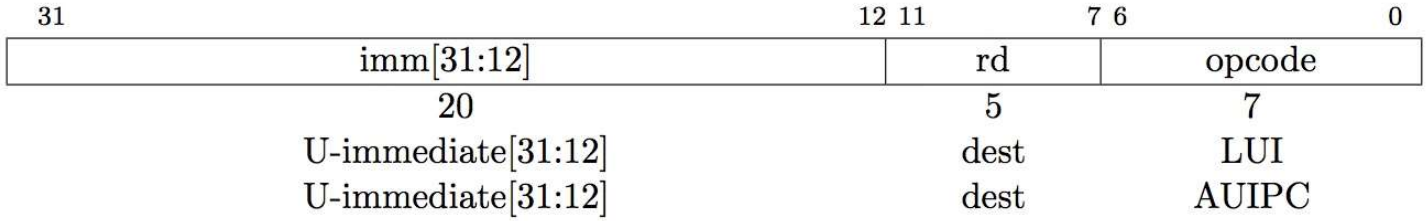
Branch Example, complete encoding

```
beq   x19,x10, offset = 16 bytes
```



imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU

U-Format for “Upper Immediate” instructions



31 — 12	11 — 7	6 — 0
imm[31:12]	rd	opcode
20	5	7
U-immediate[31:12]	dest	LUI
U-immediate[31:12]	dest	AUIPC

Has 20-bit immediate in upper 20 bits of 32-bit instruction word

One destination register, rd

Used for two instructions

- LUI – Load Upper Immediate
- AUIPC – Add Upper Immediate to PC

LUI to create long immediates

- LUI writes the upper 20 bits of the destination with the immediate value, and clears the lower 12 bits.
- Together with an ADDI to set low 12 bits, can create any 32-bit value in a register using two instructions (LUI/ADDI).

```
LUI x10, 0x87654       # x10 = 0x87654000
ADDI x10, x10, 0x321# x10 = 0x87654321
```

Exemple

Feu un programa en assemblador que carregui dos valors de una determinada posició de memòria en dos registres: a1 i a0, faci la suma i ho guardi en a2. Finalment guarda el resultat en memòria.

Feu un programa en assemblador que carregui dos valors de la memòria en dos registres. Si els valors són iguals, que faci el producte, si un és negatiu que faci la suma, si els dos són positius que faci la resta i si tots dos són negatius, que els passi a positiu i els guardi en memòria.