C14
Elemente de arhitectura procesoarelor (2)

https://www.lessons2all.com/CompArchi6.php
https://www.lessons2all.com/CompArchi4.php

https://www.tutorialspoint.com/what-is-load-store-reordering-in-computer-architecture

https://www.youtube.com/watch?v=7WV0yMzc6y8
https://www.slideshare.net/slideshow/instruction-set-architecture/127332100

https://eng.libretexts.org/Bookshelves/Computer_Science/Programming_Languages/Introduction_to_Assembly_Language_Programming%3A_From_Soup_to_Nuts%3A_ARM_Edition_(Kann)

- Atunci când se proiectează un CPU, trebuie mai întâi creată o reprezentare virtuală pentru a specifica o serie de decizii arhitecturale care trebuie luate.

1.CPU-ul va fi un CPU cu set de instrucțiuni complex (CISC) sau un CPU cu set de instrucțiuni redus (RISC)?

2. Care sunt tipurile de date acceptate, de ex., CPU-ul va accepta întregi, caractere, nr. flotante, double, etc.?

3. Cât de mari vor fi elementele de date, cum ar fi întregi și adrese? Adesea, acest lucru este legat și numit dimensiunea cuvântului pentru un CPU.

4. Formatul datelor, cum ar fi utilizarea C2 pentru valori întregi, formatul IEEE 754/85 la nr. flotante și ASCII pentru caractere. Trebuie specificat formatul pentru toate tipurile de date acceptate.

5. Câte registre vor exista și cum vor fi utilizate?

6. Cum sunt transferate datele între unitățile din CPU, numite *calea de date* a procesorului?

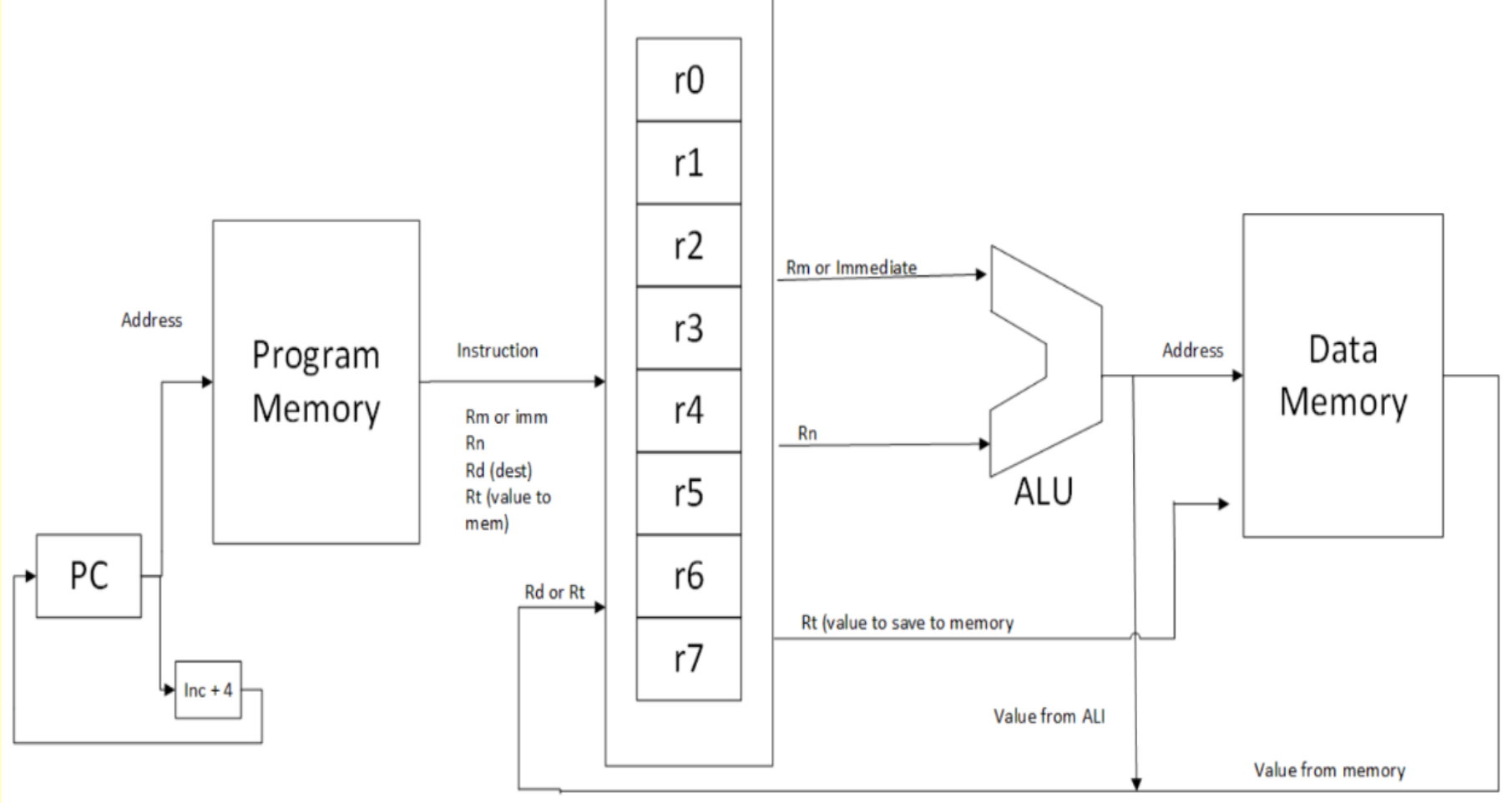
7. Cum vor fi furnizate datele către ALU sau, mai precis, computerul va avea o organizare cu 0 adrese (stivă), 1 adresă (acumulator) sau 2/3 adrese (registre generale)?

8. La proiectarea unei UCP, memoria poate fi accesată direct prin instrucțiuni sau accesul la memorie poate fi limitat la instrucțiuni *load-store*. (CPU-ARM)

9. Există o memorie unificată, numită arhitectură von Neumann/Princeton, sau memoria este împărțită între zona de cod și date, numită arhitectură Harvard.

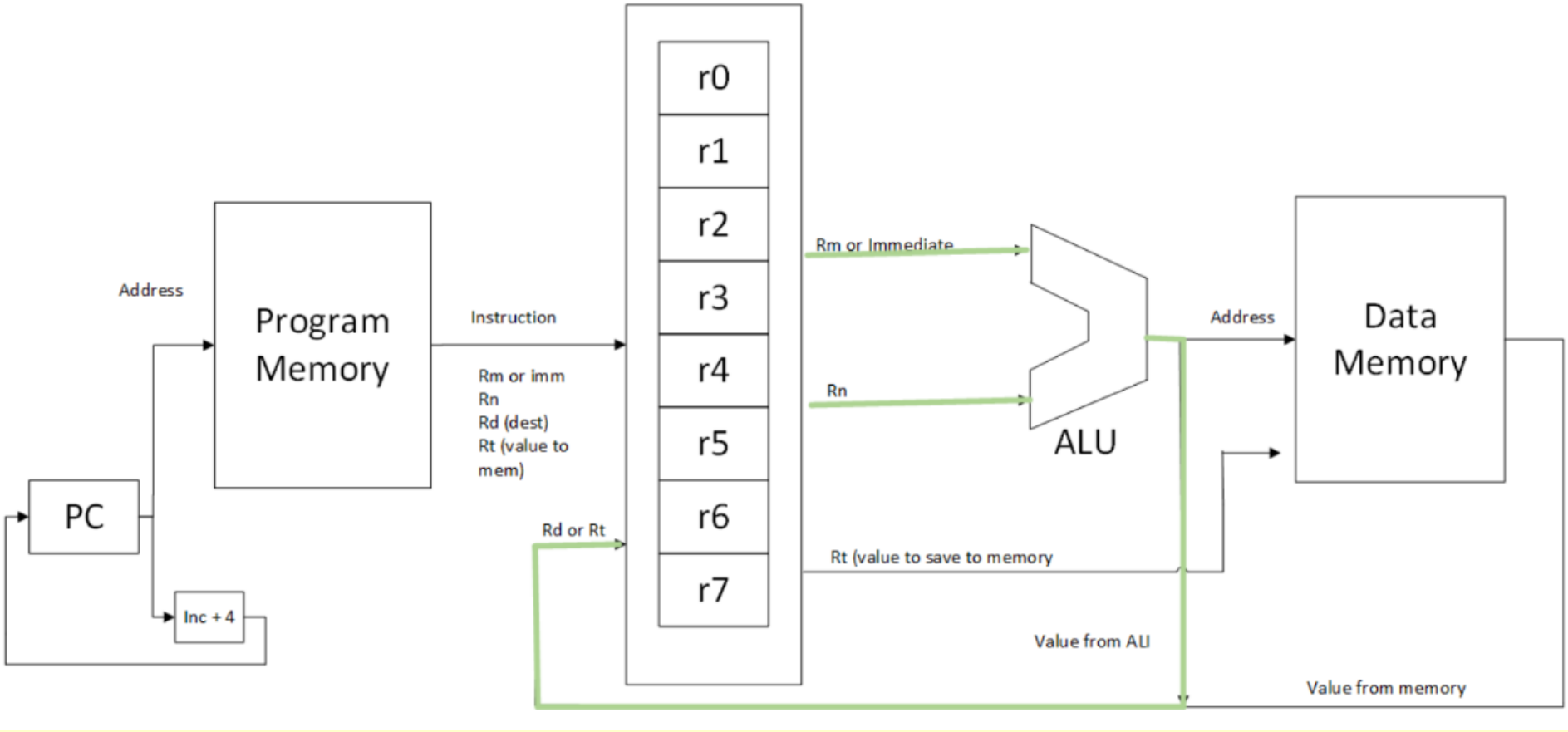
Arhitectura Load–Store

- Atunci când se proiectează o CPU , *există două moduri de bază* în care procesorul poate accesa memoria. Procesorul poate permite accesul direct la memorie ca parte a oricărei instrucțiuni sau poate permite accesul la memorie numai cu instrucțiuni speciale numite instrucțiuni de încărcare și stocare.
- Un CPU care permite accesarea memoriei de către orice instrucțiune are, de obicei, *instrucțiuni de lungimi diferite și necesită ca CPU-ul să consume mai multe cicluri de ceas* pentru decodarea instrucțiunii și accesarea memoriei. Acest lucru este mai frecvent în cazul computerelor cu set de instrucțiuni complexe (CISC), cum ar fi seria de CPU-uri Intel x86.
- ARM este un alt tip de CPU, numit *computer cu set de instrucțiuni redus* sau arhitectură RISC. Unul dintre criteriile majore de proiectare la crearea unui CPU RISC a fost *ca toate instrucțiunile să fie regulate,* ceea ce înseamnă că instrucțiunile ar avea toate aceeași dimensiune și ar necesita același timp pentru decodare și executare. Această regularitate a instrucțiunilor *permite CPU-ului să funcționeze mai rapid* și să fie optimizat folosind tehnici precum *proiectarea pipeline*, care sunt dificile, dacă nu imposibile, pe un computer CISC.
- De asemenea, permite compilatorului să profite de un număr mai mic de instrucțiuni mai regulate, în loc de un număr mare de instrucțiuni generice complexe. Posibilitatea de a compila programe folosind comportamente specifice pentru un program, în loc să folosească instrucțiuni generice complexe, permite *compilatoarelor să optimizeze mai bine codul,* făcând programele mai rapide.

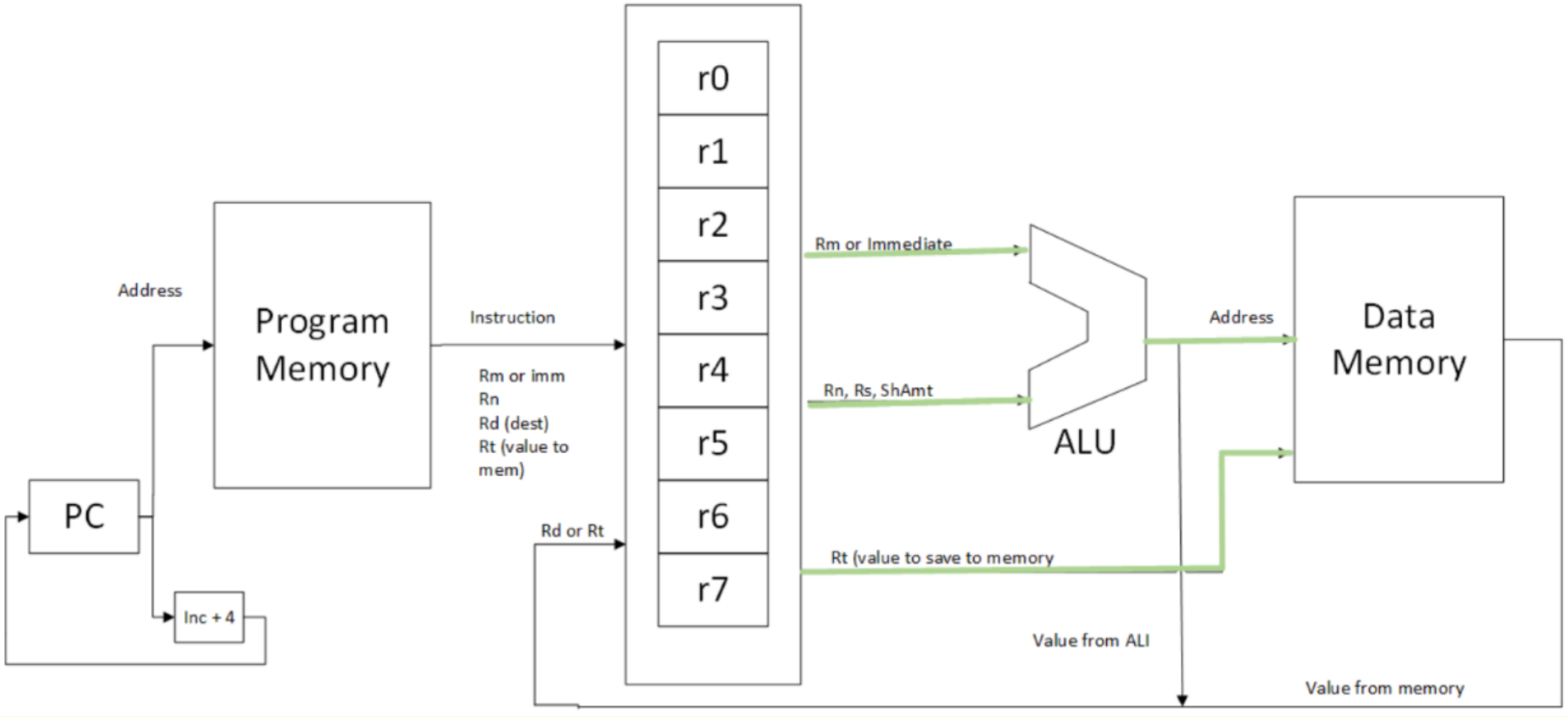


Arhitectura load-store CPU cu 3-adrese

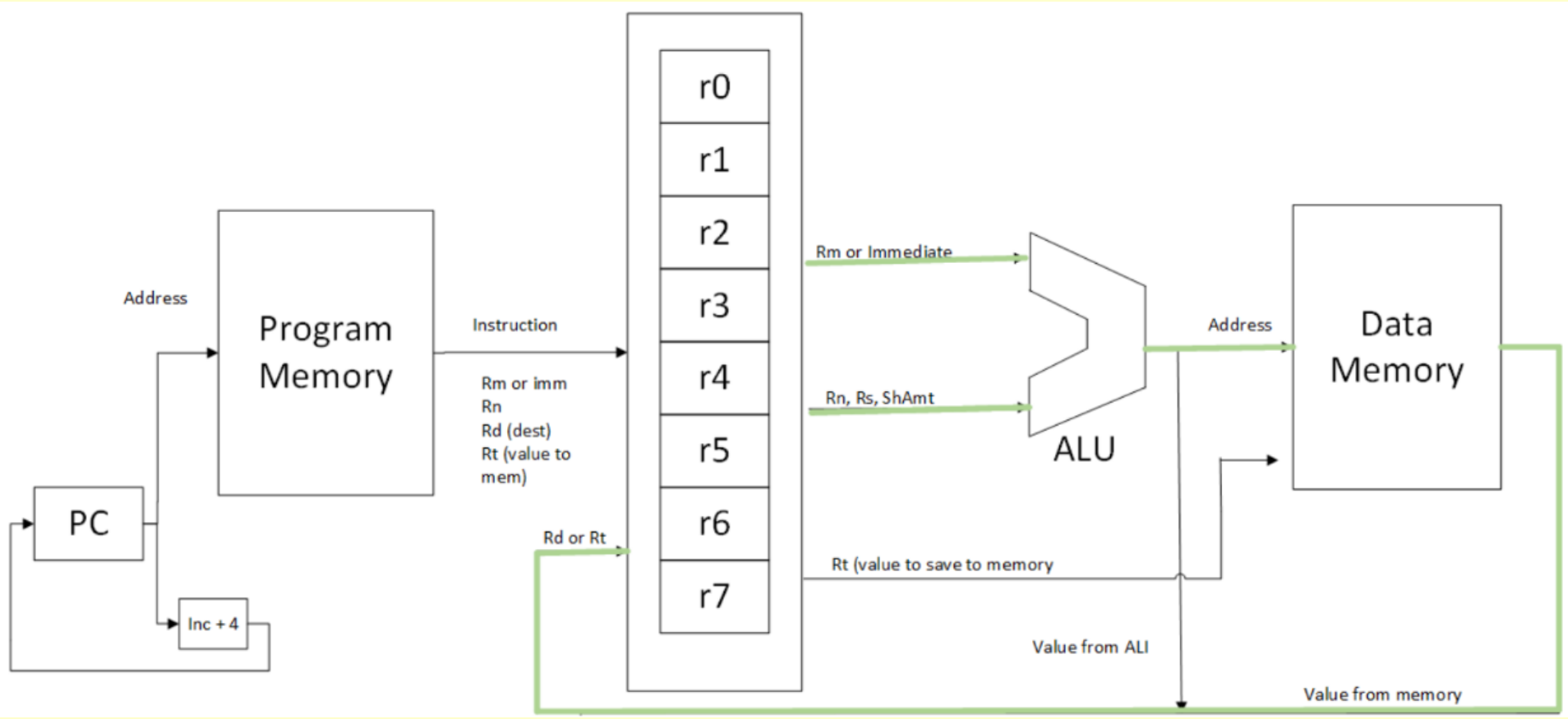
- Calculatoarele RISC nu permit tuturor instrucțiunilor să acceseze memoria, ci au operații speciale pentru încărcarea și stocarea datelor în registre. Unitățile interne ale CPU utilizează registre pentru introducerea datelor și nu pot accesa memoria direct.
- Datele externe din memorie trebuie mai întâi încărcate într-un registru înainte de a putea fi utilizate.
- Această diagramă bloc CPU este o arhitectură de încărcare și stocare cu 3 adrese.



Arhitectura load-store CPU cu 3-adrese evidentiind calea de date



Arhitectura load-store CPU cu 3-adrese evidentiind store-operation



Arhitectura load-store CPU cu 3-adrese evidentiind load-operation

```
LDR Rt, [Rn, immediate]
LDR Rt, [Rn, Rm]
STR Rt, [Rn, immediate]
STR Rt, [Rn, Rm]
```

```
LDR r1, [r2, #4]
LDR r1, [r2, r3]
STR r1, [r2, #4]
STR r1, [r2, r3]
```

Auto incrementarea registrului Rt

- Limbajul de asamblare ARM are o caracteristică utilă atunci când execută operații de încărcare și stocare; permite actualizarea automată a registrului Rn cu valoarea calculată cu adresa de memorie utilizată. Aceasta se numește auto-incrementare a registrului.
- Următoarele ex. ilustrează modul de specificare a auto-incrementării.

Fara auto-incrementare:

```
LDR r1, [r2, #4]
STR r1, [r2, r3]
```

- Fără auto-incrementare, valoarea / r2 nu se modifică la executarea unei instrucțiuni LDR sau STR.

Pre-incrementare

```
LDR r1, [r2, #4]!
STR r1, [r2, r3]!
```

- Cu pre-incrementare, valoarea/r2 este modificată înainte de calcularea adresei, iar noua adresă este utilizată la executarea unei instrucțiuni LDR sau STR.

Post-incrementare

```
LDR r1, [r2], #4
STR r1, [r2], r3
```

- Cu post-incrementare, valoarea/r2 este modificată după calcularea adresei, iar adresa veche este utilizată la executarea unei instrucțiuni LDR sau STR.

Moduri de adresare în limbajul de asamblare ARM

Diferitele moduri de adresare sau modalități de accesarea variabilelor în limbajul de asamblare ARM. Modurile de adresare care vor fi abordate sunt: imediat, direct, direct prin registru, indirect prin registru, indirect prin registru cu offset, indirect și relativ la PC.

Adresare imediată - valoarea care trebuie utilizată este inclusă în instrucțiunea însăși. Un exemplu de adresare imediată este următoarea instrucțiune ADD:

```
ADD r1, r2, #12;  r1=r2+12
```

- o valoare imediată este foarte diferită de o constantă. O valoare imediată nu este stocată în memoria de date, ci face parte din instrucțiune, astfel încât nu este necesar să încărcați valoarea într-un registru înainte de a o utiliza. O constantă este o valoare stocată în memorie, ca orice altă variabilă, cu excepția faptului că valoarea sa nu poate fi modificată. Pentru a utiliza o constantă, valoarea trebuie încărcată într-un registru, ceea ce face ca utilizarea unei constante să fie potențial mai costisitoare decât utilizarea unei valori imediate.

Adresare directa - adresa unei valori este cunoscută. de ex. atunci când valoarea este stocată la o etichetă.

```
        LDR r1, =num
        LDR r1, [r1, #0]
num: .word 5
```

În acest fragment de cod, există o adresă cunoscută a valorii care trebuie încărcată. În acest ex. *mai întâi adresa lui num (nu valoarea 5) este încărcată în registrul r1*. Adresa din registru este apoi utilizată pentru a încărca valoarea în r1.

Adresare directă la registru - O valoare este o adresare directă la registru dacă valoarea este stocată în registru. În instrucțiunea următoare, adresarea directă la registru este utilizată pentru ambele componente ale adunării.

```
ADD r1, r2, r3        ; r1=r2+r3
```

Adresre la Registru indirect - adresa variabilei care trebuie utilizată este stocată în registru, iar valoarea de la acea adresă este încărcată într-un registru pentru a fi utilizată ulterior.

- Acest lucru poate fi util în mai multe situații pentru a accesa date.
  De ex., următoarea ar fi o modalitate eficientă de a accesa *un șir de date întregi* alocate începând de la adresa etichetei arr. În acest caz, r1 va fi incrementat la valoarea următorului element pe măsură ce fiecare valoare a elementului șirului este încărcată în r2.

```
LDR r1, =arr        ;r1 =adr. array
LDR r2, [r1], #4    ;r2 = continutul adresei (r1) si r1=r1+4
.data
      arr: .word 10
```

Adresare la registru indirect cu offset - este cel mai eficient mod de a gestiona adresarea atunci când există o adresă de bază și valori care se află la un offset cunoscut al acelei baze.

Un ex. sunt structurile sau clasele. Pentru a vedea acest lucru, luam în considerare următoarea clasă Java:

```
class A {
    long a;
    int b;
}
```

- Pentru o adresă care indică începutul clasei în r1, valoarea pentru fiecare variabilă poate fi încărcată în r2 folosind următoarele instrucțiuni:

```
LDR r2, [r1, #0]
LDR r3, [r1, #8]
```

Adresarea indirectă este similară cu adresarea indirectă prin registru, cu excepția faptului că valoarea din memorie este de fapt o adresă către o altă valoare.

- De fapt, valoarea adresată poate fi ea însăși o adresă. De exemplu: luați în considerare următorul fragment de cod:

```
        LDR r1, =addr_num
        LDR r1, [r1, #0]
        LDR r1, [r1, #0
.data   num: .word 5
        addr_num: .word num
```

În acest fragment de cod, *valoarea de la adresa etichetei addr_num* este *adresa (referința la) adresa etichetei num*. Pentru a găsi valoarea reală, lanțul de adrese/ lanțul de referințe trebuie parcurs până când se găsește valoarea finală.
Acest lucru devine interesant deoarece oferă o perspectivă asupra relației dintre referințe și valori. *O valoare reprezinta datele de la adresa unei referințe.*
Referințele indică valori, iar valoarea depinde de relația sa. Acest lucru duce la definirea a doi termeni noi: *un tip de referință și un tip de valoare*.
Un tip de referință implică faptul că valoarea de la adresă este o referință la o altă valoare.
Un tip de valoare este o valoare finală în lanțul de valori de referință și este o din program.

Adresare relativa

PC-ul conține adresa instrucțiunii care trebuie executată, astfel cand avem o ramificare într-un program, implică modificarea PC-ului la o nouă valoare. Cum se calculează noua valoare care va fi utilizată pentru PC? Pentru a face acest lucru, vor fi introduse două tipuri de adresare: ***adresarea relativă la PC și adresarea absolută.***

1. *O adresă absolută este o adresă absolută în memorie.*
   - Daca adresa memoriei instrucțiunii de executat este 0x35fc. Pentru a ramifica la această adresă, tot ce ar fi necesar este ca PC-ul să fie setat la această valoare (de exemplu, MOV pc, #0x35fc),

iar codul ar începe să se execute la noua adresă. Adresarea absolută este ușor de înțeles, dar este oarecum dificil de implementat. La compilarea și asamblarea unui program, adresele absolute ale instrucțiunilor mașinii nu sunt încă cunoscute. Adresa absolută a instrucțiunii este atribuită la momentul link-editarii programului și, prin urmare, calculul adreselor trebuie amânat până la momentul link-editarii.

2. Adresarea relativă la PC

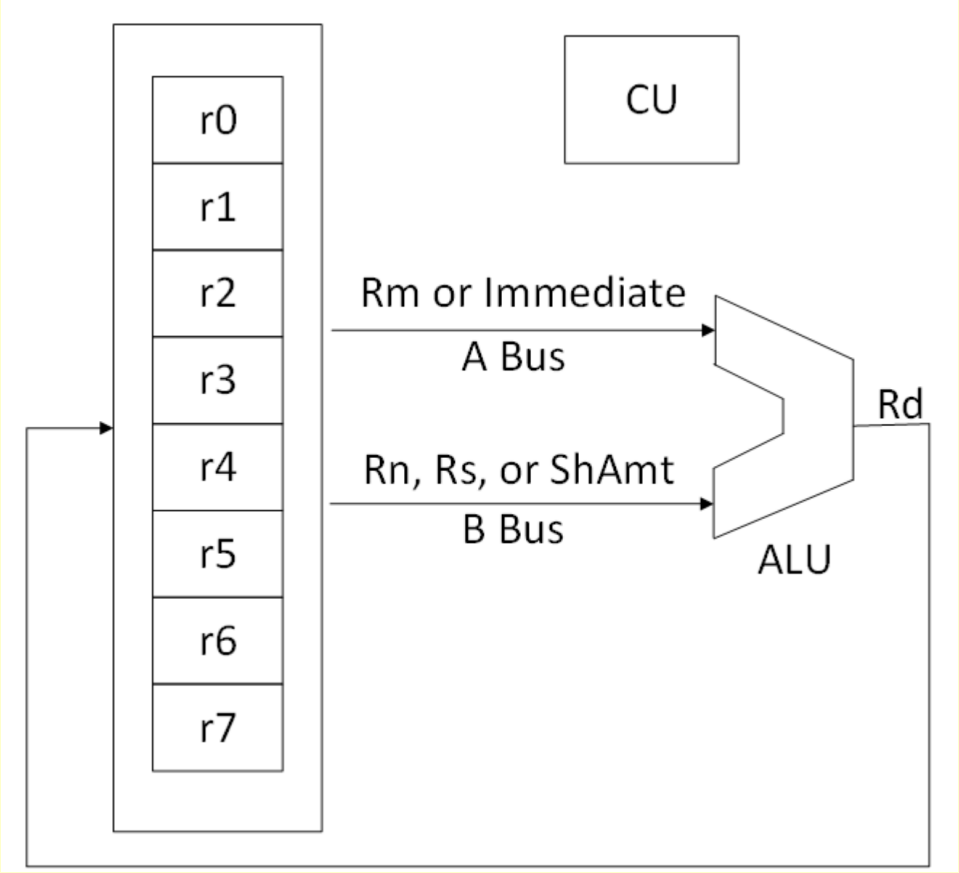
- Atunci când se execută un program, adresa absolută a PC-ului este cunoscută. Dacă se cunoaște și distanța de la PC la o altă instrucțiune, adresa instrucțiunii către care se face ramificarea poate fi calculată prin adăugarea distanței respective la PC, iar adresa absolută a acelei instrucțiuni poate fi calculată. Acest lucru este cunoscut sub numele de ***adresare relativă la PC.***

În adresarea relativă pe PC, asamblorul sau compilatorul poate calcula distanța dintre orice instrucțiune și orice altă instrucțiune din același fișier.

De exemplu. Instrucțiunea „B label_1” intenționează să ramifice la label_1. Instrucțiunea de ramificare transmite *distanța instrucțiunii de ramificare la label_1* către CPU, iar CPU calculează și ramifică la acea instrucțiune. Înainte de a arăta cum se calculează o ramificare, există câteva considerații importante pentru calcularea ramificărilor.

Obs. În primul rând, deoarece instrucțiunile au toate o dimensiune de 4 octeți, fiecare instrucțiune se află la 4 octeți distanță de instrucțiunea anterioară, așa că ați putea crede că fiecare instrucțiune ar adăuga 4 la valoarea distanței pentru ramificare. Cu toate acestea, deoarece instrucțiunile sunt aliniate la nivel de cuvânt, ultimele două zerouri sunt eliminate din valoarea distanței. Aceasta înseamnă că distanța utilizată în instrucțiunea de ramificare poate fi obținută prin numărarea numărului de instrucțiuni dintre instrucțiunea de ramificare și instrucțiunea către care se ramifică. Rețineți că acest truc funcționează, dar nu este reprezentativ pentru ceea ce se întâmplă în realitate.

În al doilea rând, când ramificația este calculată efectiv, valoarea din PC este PC-ul instrucțiunii de ramificare plus 8. Deci, când executați ramificația, ar trebui să începeți să numărați de la instrucțiunea cu două înaintea instrucțiunii de ramificare curente atunci când calculați distanța de ramificare. Astfel, în acest exemplu, PC-ul la instrucțiunea „B label_1” este 0x35e8, dar distanța utilizată în instrucțiunea de ramificare este 3 (0x35e8 + 8 + 3*4 = 0x35fc).



- În continuare, liniile cu săgeată din diagramă sunt de fapt 32 de fire/bus, fiecare transportând un bit. Setul superior de fire se numește magistrala A, iar cel inferior de la se numește magistrala B.
- Din punctul de vedere al programatorului in asamblare, există două formate de bază ptr. instrucțiuni. Primul format utilizează *un registru de destinație și două registre sursă*

OP R_d, R_1, R_2 // inseamna $R_d \leftarrow R_1$ OP R_2 ←OP este operatorul (+,*,-....) ex. ADD r1, r2, r4

Al doilea format utilizează un *registru de destinație, un registru sursa și o valoare imediată*.

OP R_d, R_1, #num //inseamna $R_d \leftarrow R_1$ OP #num ex. ADD r1, r2, #87 ; LSL r1, r1, #2

Instrucțiuni cu 3-adrese (Exemple)

Instructiuni MOV

MOV Rd, immediate; MOV r1, #36 ; r1=36

MOV Rd, Rm ; MOV r1, r2 ; r1 ← r2

Instructiuni ADD / SUB

ADD Rd, Rn, immediate; ADD r1, r2, #36 ; r1 ← r2 + 36
SUB Rd, Rn, immediate; SUB r1, r2, #36 ; r1 ← r2 – 36
ADD r1, r2, r3 ; r1 ← r2 + r3
SUB r1, r2, r3 ; r1 ← r2 - r3

Instructions MUL, SDIV, and UDIV

MUL Rd, Rm, Rs ; MUL r1, r2, r3
SDIV Rd, Rn, Rm ; SDIV r1, r2, r3
UDIV Rd, Rn, Rm ; UDIV r1, r2, r3

Instructiuni Logice : AND, OR, XOR, and BIC

```
MOV r1, #0x42   // put a character ‘B’ in r1
 MOV r2, #0x20   // bit-mask for upper to lower case
 ORR r1, r1, r2  // r1 now contains the character ‘b’
       MOV r1, #0x63   // put a character ‘c’ in r1
       MOV r2, #0x20
       BIC r1, r1, r2          //r1=0x43; r1=“C”
```

https://eng.libretexts.org/Bookshelves/Computer_Science/Programming_Languages/Introduction_to_Assembly_Language_Programming:_From_Soup_to_Nuts:_ARM_Edition_(Kann)/04:_New_Page/4.04:_New_Page