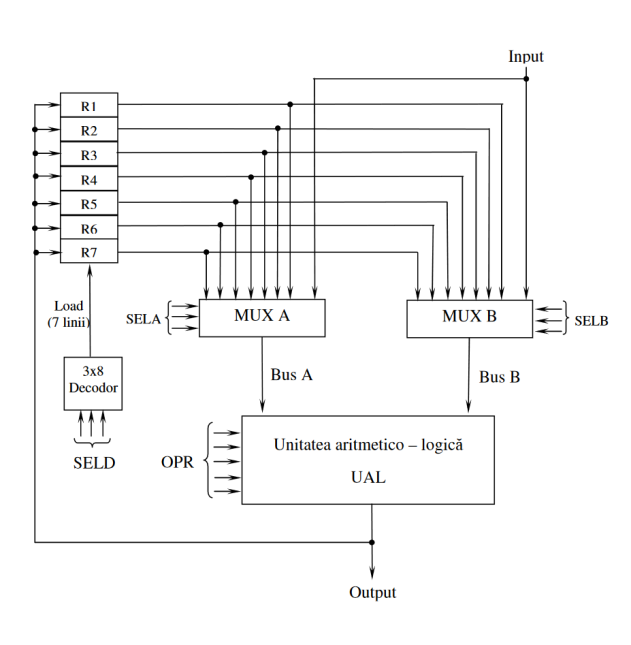
**PROIECT ASC – IMPLEMENTARE DATA PATH + UAL**

**1. Introducere**

Proiectul prevede implementarea Unității Aritmeticologice și a Datapath-ului prezente în arhitectura unui procesor. Prin lipsa unei Unități de Comandă și Control și a RAM-ului am fost nevoiți să implementăm un interpretor care preia comenzi de la utilizator cu sintaxa similară limbajului de asamblare (ex. ADD 3 4 – adună conținutul registrului 4 la registrul 3) pentru a prezenta și partea practică a programului. De asemenea, programul dispune și de o listă de regiștrii care acționează ca memoria pe care lucrăm.

 Interpretorul nostru transformă inputul utilizatorului într-o secvență de biți pe care datapath-ul și UAL lucrează pentru a executa operații direct pe regiștrii pe care-i reținem.

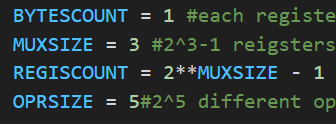
Noi am ales să desfășurăm acest proiect nu numai pentru a ne împrospăta cunoștințele de programare în python dar și pentru a avea o înțelegere mai bună asupra arhitecturii procesorului și a modului în care acesta face calcule. La redactarea codului, am decis să nu folosim librării precum Numpy pentru lucrul cu array-uri și Struct pentru a ne ușura lucrul cu bytes. Am vrut să ne oferim provocarea de a scrie totul în python nativ pentru a ne obișnui cu sintaxa nefamiliară a limbajului.

Drept documentatie ne-am inspirat din materialele oferite la cursul de Arhitectură a sistemelor de calcul, mai ales de schema (1).

1. **schema functionalitatii Datapath-UAL)**

În următoarea secțiune vom prezenta aprofundat funcționarea Datapath-ului și a UAL-ului.

**2. Implementare**

**2.1 Variabile și structură** – Shaikh Saud

Pentru a ne ușura legarea bucăților la care a lucrat fiecare și pentru a oferi o anumită scalabilitate algortimilor ne-am decis

la un set de variabile și constante globale pe care să lucrăm. Constantele globale sunt:

- BYTESCOUNT care oferă numărul de bytes prezenți într-un registru (în exemplul nostru 1 Byte 🡪 8 biți).

- MUXSIZE reprezintă numărul de select bits ale multiplexoarelor MUXA și MUXB și decodorului SELD (numărul întreg a cărei reprezentare în binar este conținutul acestora reprezintă indicele registrului din vectorul de regiștrii cu care vom lucra). Implicit obținem astfel numărul de regiștrii care vor acționa ca memorie principală a programului (în număr de 2^MUXSIZE-1 pentru a putea lucra cu inputul de la utilizator și outputul care se va face în consolă).

- OPRSIZE determină numărul de select bits ai OPR prin care Unitatea Aritmeticologică decide ce instrucțiune va face. De asemenea numărul de instruțiuni suportate se face implicit, acesta fiind de 2^OPRSIZE.

Pe lângă aceste constante mai dispunem și de SELA, SELB, SELD, OPR și o listă de regiștri de care am menționat mai sus. Vom reține și o variabilă output în caz în care vrem să afișăm în consolă conținutul unui registru (folosit în mare parte pentru testarea functionalității algoritmilor).

Pentru a ne ușura treaba am decis să avem un dicționar tip (Operație: codul respectiv în biți) cu toate operațiile de care dispune UAL. Inițializările tuturor acestor variabile se regăsește în funcția **initializeRegisters()** din modulul **variables.py**.

**2.2 Unitatea Aritmeticologică** – Ureche Tudor

Unitatea aritmetico-logică se ocupă cu efectuarea operatiilor logice și aritmetice desemnate de codul de operatie (operation code) dintre 2 termeni (A și B).

Codul de operație și termenii sunt preluați de la funcția interpretor de catre funcția ALU ca parametri. Variabila opr este pe 5 biți și reprezinta operation code-ul, ceea ce rezultă că putem efectua maxim 2^5-1 operații. Variabilele a și b pot avea câți biți dorim, unitatea aritmetico-logică poate efectua operații pe orice număr de biți.

Ca și operații notabile se enumeră înmulțirea și împărțirea:

**2.2.1 Înmulțirea**

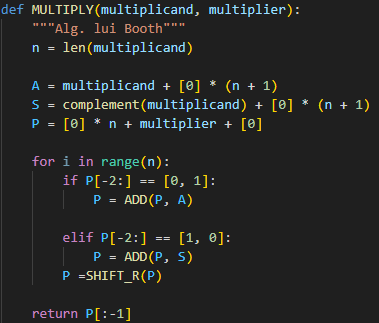
Înmulțirea se bazează pe **Algoritmul lui Booth**. Avem 3 regiștri (A, S și P) pe baza cărora se face înmulțirea.

🡪n = lungimea deînmulțitului

🡪A = deînmulțitul+ (n+1) zerouri în coadă

🡪S = complementul deînmulțitului+(n+1) zerouri în coadă

🡪P = n zerouri + înmulțitorul + un zero în coadă

Algoritmul lui Booth este implementat ca o serie de adunări condiționate în registrul P.

Se iau în considerare ultimii doi biți ai lui P:

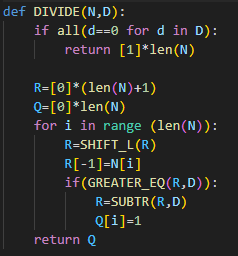
* Pentru 00 sau 11 se face shiftare la dreapta;
* Pentru 01 se adună în P registrul A;
* Pentru 10 se adună în P registrul S.

**(2. snippet cu algoritmul de înmulțire)**

De menționat că în urma oricărei adunări garantat ultimii 2 biți vor fi identici => shiftare obligatorie. Acest procedeu se repetă de n ori.

Rezultatul final va fi stocat în registrul P, de la care omitem bitul cel mai semnificativ (cel mai din stânga).

**2.2.2 Împărțirea**

Algoritmul de împarțire este inversul algoritmului de înmulțire. Se bazează pe scăderi repetate și shiftări la stânga.

🡪Inițial se verifică dacă deîmpărțitul (variabila D) este 0. În caz pozitiv, se returnează un registru plin de zerouri de lungimea deimpărțitului.

🡪Se consideră R registrul pentru rest, inițial plin cu zerouri. Acesta este mai lung cu un bit pentru operația de shiftare.

🡪Mai avem registrul Q de lungimea deîmpărțitului în care vom stoca rezultatul final.

**(3. snippet cu algoritmul de impartire)**

Cum funcționează:

* Parcurgem deîmpărțitul de la stânga la dreapta
* Shiftăm la stânga restul și înlocuim bitul cel mai semnificativ al restului cu bitul pozitiei curente din deîmpărțit
* În cazul în care restul ajunge mai mare decat împărțitorul, se scade împărțitorul din rest iar bitul curent din rezultat devine 1

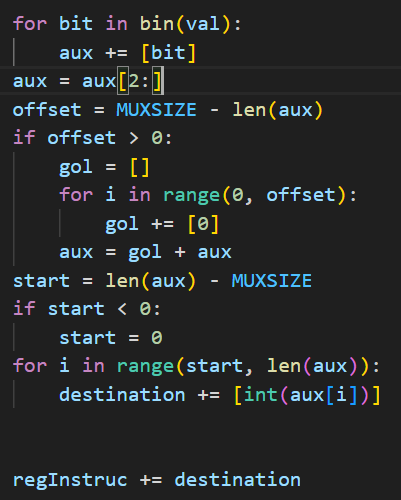
**2.3 Data Path-ul**– Shaikh Saud, Tudose Alexandru

Ne folosim de funcțiile din modulul ALU și variabilele din variables pentru a ne ușura treaba de a lega toate functiile între ele, formand astfel o ierarhie între module.

**2.3.1 Interpretorul** **–** Shaikh Saud

După cum am precizat în introducerea documentului, pentru a arăta funționalitatea programului și din lipsa UCC am scris un interpretor care preia datele de la tastatură. Inputul dat de utilizator trebuie să fie sub forma [COMANDA LOCATIE1 LOCATIE2], LOCATIE1 fiind sursa și mai ales destinația rezultatului comenzii și LOCATIE2 sursa. Din conveniență secțiunea comandă va fi scrisă cu litere mari.

În marea majoritate a cazurilor, rezultatul operației se va pune în LOCATIE1. Unul din cazurile speciale este determinat de comanda “OUT”. În acest caz, LOCATIE1 va acționa doar ca sursă, destinația fiind consola. Un alt caz special este pentru comanda “IN”. În această privință, LOCATIE2 nu va mai prezenta un index din vectorul de regiștrii, ci inputul de la utilizator.

În rest, algoritmul de transformare este simplu: din scrierea binară a numerelor LOCATIE1 respectiv LOCATIE2 vom lua ultimii MUXSIZE biți și pe restul îi vom face 0 pentru a oferi scalabilitate și acestei părți.

Rezultatul conversiunii lui LOCATIE1 se va pune în registrul destination, iar al LOCATIE2 în sursa în care vor fi adăugate registrului de instrucțiuni. Dacă comanda este ”IN” registrul sursa va fi umplut cu 1 și reprezentarea binară a LOCATIE1 va fi reținută pe 8\*BYTESCOUNT biți pentru a fi utilizată ulterior în UAL. O variabilă callInput va fi folosită panetru a semnala acest fapt. În cazul în care comanda este ”Out” registrul destinație la final în loc să fie în original copiat la capăt va fi reconstruit și umplut cu 1, semnalând că afișarea se va face în consolă.

**(4. snippet cu alg pentru destinatie)**

**2.3.2 Pipeline-ul** **–** Tudose Alexandru

După traducerea inputului în date pe care le putem prelucra (regiștrii – vectori de biți), acestea vor trece printr-o succesiune de 4 etape care constituie pipeline-ul procesorului: FETCH, DECODE, EXECUTE și LOAD; fiecare cu rolul ei specific. Am decis să le împărțim în 4 funcții separate pentru a evidenția funcționalitatea acestora.

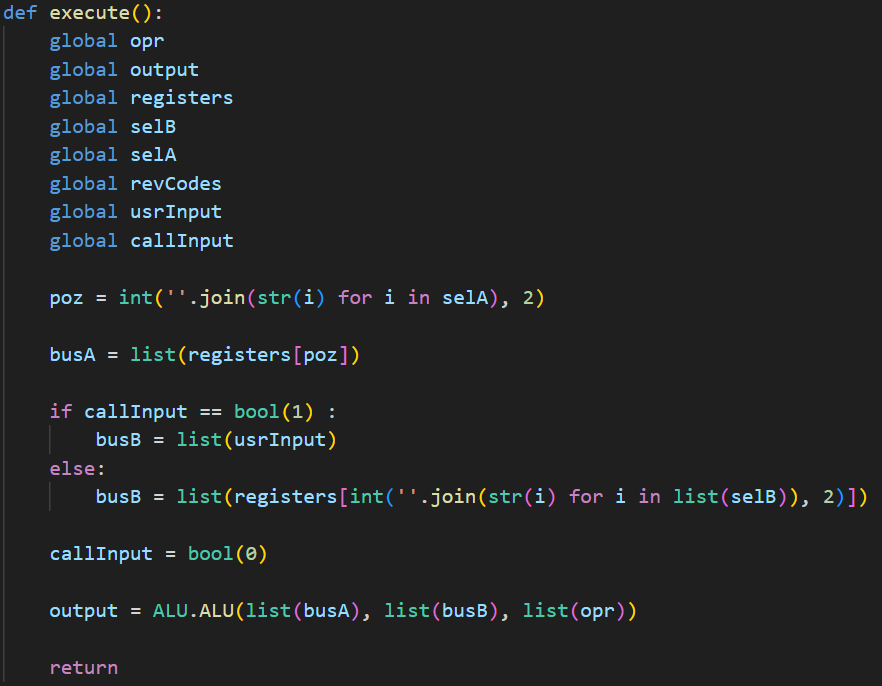
FETCH preia după un parametru K o succesiune de OPRSIZE+3\*MUXSIZE biți din registrul de instrucțiune pentru a fi folosiți în continuare. Pe baza acestora, în DECODE vom împărții această listă în funcție de formatul de instrucțiune pe care ni-l definim prin mărimea la OPRSIZE și MUXSIZE și a cuvântului de control de forma |OPR|SELA|SELB|SELD|.

EXECUTE prevede umplerea busA respectiv busB cu datele care vor fi prelucrate în Unitatea Aritmeticologica. Toate datele vor fi pasate ca obiect nou pentru a nu lucra direct cu ele și astfel

schimbă conținutul regiștrilor din greșeală. Pentru a ști ce regiștrii vrem să încarcăm în busA, respectiv busB, vom converti conținutul lui SELA și SELB din binar în baza 10 prin

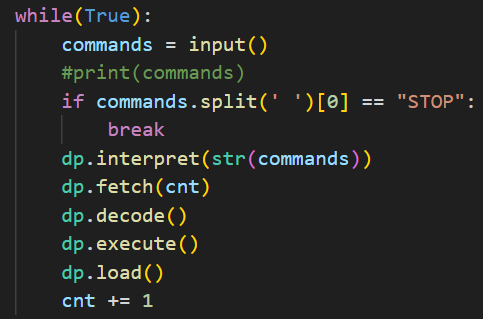
**int(''.join(str(i) for i in selA), 2)**

imitând astfel funcționalitatea unui multiplexor. UAL va fi chemat pe datele specifice, rezultatul punând-ul în output care va fi folosit în ultima etapă.



Ultima etapă, LOAD, va încărca rezultatul în funcție de SELD undeva în lista de regiștrii sau în consolă. Prin toate aceste etape, reușim să interpretăm modul de bază prin care procesorul lucreză cu date.

**(5. snippet cu algoritmul de executare a datelor)**

**2.4 Testare**– Shaikh Saud

În final, pentru a evidenția partea practica a codului am decis să scriem și un modul simplu de testare: într-o buclă infinită așteptăm comanda de la utilizator. Dacă aceasta nu conține “STOP”, inputul va trece prin data path și astfel cele 4 etape ale pipeline-ului. Altfel, programul se va opri cu totul. Pe lângă toate acestea modulul ne-a oferit și posibilitatea de a testa funcționalitatea programului pe parcursul redactării codului.

**(6. snippet cu cod din modulul de testare)**

****

**(7. exemplu prezentând conținuturile regiștrilor 4 și 5 și folosirea operațiilor OUT, IN și ADD)**

**3. Probleme depășite**

Pentru toți o problemă mare o constituie sintaxa și modul de gândire al limbajului python. Trecerea de la C++ intensiv la python fără vreo anumita experiență cu limbajul în trecut ne-a lungit puțin timpul total de lucru.

Shaikh Saud: interpretorul a fost o mică provocare pentru mine, totul trebuind să meargă pe majoritatea inputurilor și să poată funcționa și pe orice mărime a setului de date. Cu toatea acestea încă este incomplet: nu prezintă suport pentru comenzi sub forma OPERATIE-LOCATIE, este impus ca toate comenzile să fie scrise cu litere mari și ca regiștrii și numerele să fie în intervalele lor specifice astfel încât programul să nu crape. O altă problemă întâmpinată tot în interpretor este legată de comanda “IN”: programul salvează rezultatul într-o variabilă care va fi ulterior folosită în UAL – acest lucru nu ne permite să scriem un set de comenzi conținând mai multe “IN”-uri pe care să le prelucrăm, variabila fiind constant suprascrisă la momentul construirii registrului de instructiuni.

Tudose Alexandru: nu văd să fi avut specific probleme în redactarea codului, mai mult m-am găsit blocat la înțelegerea tematicii pe care am dorit să o acoperim.

Ureche Tudor: operațiile pe orice număr de biți au fost dificil de implementat, deoarce necesită gândirea operațiilor la general, de aceea algoritmii mai complecși de operație au prezentat probleme pe aproape toată durata implementării proiectului (în special algoritmii de înmulțire și împărțire).

**4. Concluzie și dezvoltări ulterioare**

În urma dezvoltarii într-o perioadă de timp oarecum scurtă a acestui proiect am câștigat câteva cunoștințe despre procesoare, o tematică interesantă dar greu de parcurs din punctul nostru de vedere și python, un limbaj cu care sigur ne vom mai intersecta în viitor.

Dezvoltări ulterioare ar cuprinde adăugarea unei interfețe grafice, transpunerea la un procesor intreg prin adaugarea unei unități de comandă și control și implementarea unei memorii RAM, finisarea interpretorului astfel încât să nu produca erori în mijlocul rularii, rescrierea codului să nu mai fie la fel de “messy” și mai object-oriented.

**5. Bibliografie**

1. Wikipedia - Booth's Multiplication Algorithm - <https://en.wikipedia.org/wiki/Booth's_multiplication_algorithm> [Accessed 25.5.2024]
2. Wikipedia - Division Algorithm: Integer division(unsigned) with remainder - <https://en.wikipedia.org/wiki/Division_algorithm#Integer_division_(unsigned)_with_remainder> [Accessed 26.5.2024]
3. Stack Overflow – pentru transformarea a unei liste de 0 și 1 într-un număr întreg a cărui reprezentare binară este conținutul listei - <https://stackoverflow.com/a/13659153> [Accessed 25.5.2024]