Tema 2

NOT SO SIMPLE ALU

MIULESCU CRISTINA-MARIA | 333 AB
ARHITECTURA CALCULATOARELOR

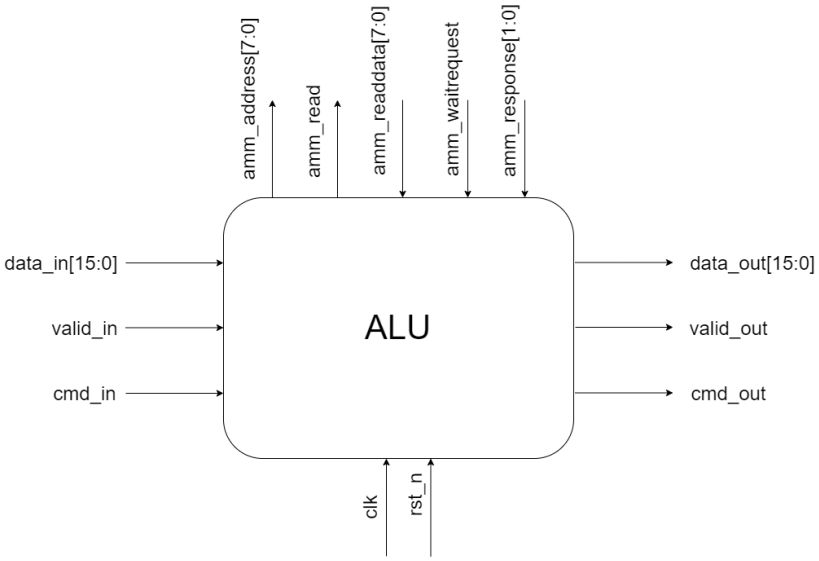
Cerinta temei

In cadrul acestei teme, am avut de implementat o unitate aritmetica logica, capabila sa execute 10 operatii.

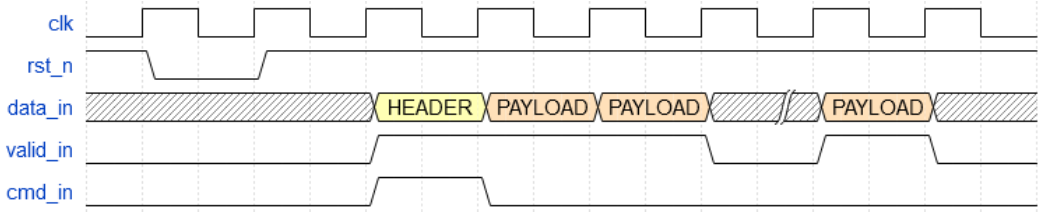
> DISCLAIMER: Desi implementarea mea nu a reusit sa respecte cerintele, voi incerca sa explic cat mai detaliat cum am gandit pasii acestei teme precum si obstacolele intampinate pe parcurs.

Modul de rezolvare

La intrare, modulul primea informatii legate de header si payload. Headerul primea informatii referitoare la codul operatie ce avea sa fie executata precum si numarul de operanzi (acestia fiind intre 0 si 63). Informatiile despre operanzi se aflau in payload unde, in functie de modul de adresare (direct / indirect), aveam sa extragem operanzii fie din payload, fie citind din memorie prin interfata AMM.



Astfel, pentru a parcurge toti pasii de decodificare a headerului si payloadurilor, de executare a operatiilor cerute si de asignare a semnalelor de iesire, am incercat sa construiesc un automat care sa respecte conditiile impuse.



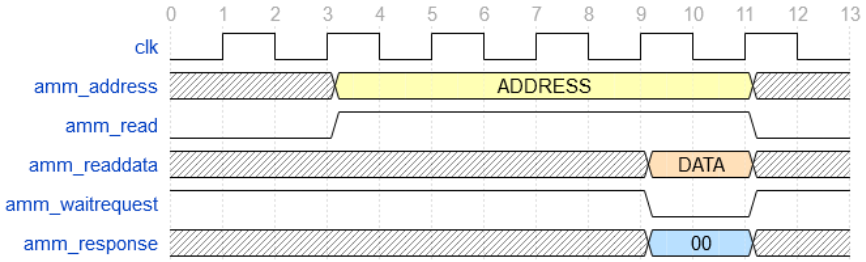
La primul pas, am implementat starea **`GET_HEADER** (probabil ca as fi putut incepe si cu o stare de `RESET in care as fi putut pune next_state = `GET_HEADER), unde atat timp cat **valid_in = 1** si **cmd_in = 1** , datele de intrare (data_in) ofereau informatii despre header. Astfel, am salvat in variabila **header_reg** pe **data_in** si totodata, l-am setat pe **count = 0** care ar reprezenta numarul de payloaduri ce ar trebui sa fie extrase in continuare. Totodata, am salvat numarul de operanzi si codul operatiei in variabilele **nof_operands** si **opcode**.

In momentul in care **nof_operands != 0**, treceam la starea `GET_PAYLOAD. In caz contrar, asignam variabilei **err** valoarea 1 si urmatoarea starea avea sa fie `RESULT_HEADER unde aveam sa ofer informatii despre output.

Urmatoarea stare implementata este **`GET_PAYLOAD** unde, atat timp cat **valid_in = 1** si **cmd_in = 0**, datele de intrare, la fiecare nou ciclu de ceas, ofereau informatii despre cate un payload. Astfel, am retinut intr-un vector **payload_reg** pe **data_in**. Prin urmare, de fiecare data cand conditia de mai sus era indeplinita, salvam pe data_in in **payload_reg[count]** (count initial 0 de la starea anterioara) si incrementam cu 1 pe count.

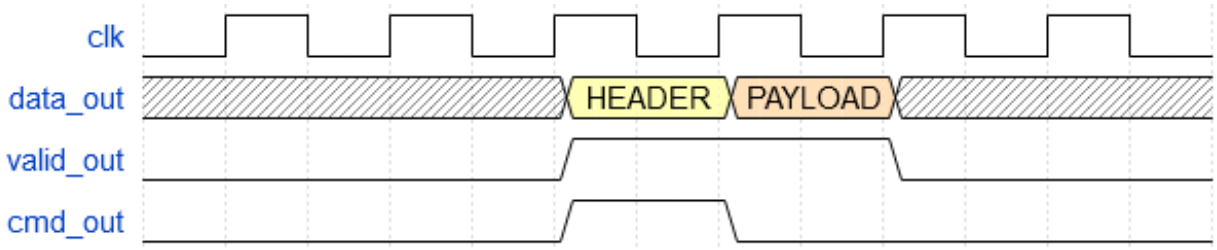
OBS: cand treceam din aceasta stare la urmatoarea (`DECODE), am observat in simulator ca nu am reusit sa imi salvez datele de intrare in header_reg si payload_reg. Probabil s-a intamplat asa deoarece nu am folosit indexarea corecta(folosind count<= count_next). Cu toate acestea, cand nu treceam la o stare urmatoare si foloseam **$display**, datele de intrare reusisera sa fie citite si se aflau in header_reg si payload_reg.

In starea de **`DECODE**, am folosit un for pentru a parcurge vectorul payload_reg. Astfel, am evaluat bitii 9:8 (modul de adresare) pentru a decide cum pot sa obtin operanzii de care am nevoie pentru executia operatiilor. In cazul in care aveam adresare directa (2'b00), atunci salvam in **op[i]** bitii de la 7:0 ai lui **payload_reg[i]** si treceam la urmatoarea stare `OPERATION. In cazul in care aveam adresare indirecta ( 2'b01), atunci asertam pe 1 pe **amm_read_reg** si salvam in **amm_address_reg** ultimii 8 biti ai lui payload_reg[i], dupa care treceam la starea `GET_OPERAND.



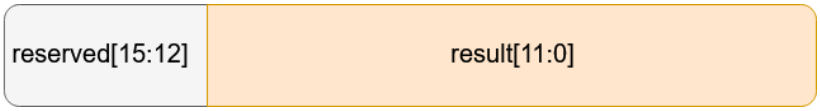
In starea **`GET_OPERAND**, din cate am inteles trebuia sa astept **minim 3 ciclii de ceas** pentru ca tranzactia sa se poata efectua. Astfel, cat timp **amm_waitrequest = 1**, reveneam la aceeasi stare curenta (`GET_OPERAND). In caz contrar, verificam daca **amm_response = 2'b00** (tranzactie incheiata cu succes) si salvam amm_readdata in op[i] dupa care treceam la starea `OPERATION. Daca **amm_response** era diferit de 2'b00, atunci setam **err = 1** si treceam la starea `RESULT_HEADER.

In starea **`OPERATION**, am analizat opcode-ul. Cu exceptia operatiei de adunare, unde rezultatul este pe 12 biti, restul operatiilor au rezultatul pe 8 biti. In cazul operatiilor `ADD, `AND, `OR, `XOR am folosit un for pentru a parcurge toti operanzii. In cazul operatiilor `NOT, `INC, `DEC, `NEG, `SHL si `SHR, a fost nevoie sa verific daca **nof_operands == 1,** in caz contrar, err = 1. Dupa ce s-a efectuat operatia respectiva, urmatoarea stare este `RESULT_HEADER.



In starea **`RESULT_HEADER,** setam pe **valid_out = 1** si pe **cmd_out = 1** iar, daca **err = 1** , pe bitul 4 din data_out_reg il punem pe 1, in caz contrar pe 0. Pe ultimii 4 biti din data_out_reg punem opcode-ul. Din aceasta stare, trecem la starea `RESULT_PAYLOAD.

In starea **`RESULT_PAYLOAD,** setam **valid_out = 1** si pe **cmd_out = 0**, iar, daca **err = 1,** pe **data_out_reg** pun valoarea **16'hbad,** in caz contrar, in functie de lungimea rezultatului operatiei efectuate, in **data_out_reg**[11:0/7:0] voi salva **result**. Astfel testul este complet si pot trece la urmatorul : `GET_HEADER.



La final, dupa ce am iesit din blocul always@, am asignat lui valid_out, cmd_out si data_out pe valid_out_reg, cmd_out_reg si data_out_reg.

Am inserat mai jos o schema a automatului implementat de mine.

SCHEMA AUTOMATULUI NOT SO SIMPLE ALU