Ingineria sistemelor
2022

PROIECTAREA ȘI SIMULAREA UNUI PROCESOR MULTI-CORE CU ARHITECTURĂ SUBLEQ

Candidat: Daniel-Bogdan Horț

Coordonator științific: Ș.l.dr.ing. Eugen Gurban

Sesiunea 2022

REZUMAT

Subleq este un limbaj ezoteric de asamblare și o arhitectură de procesor din categoria One Instruction Set Computer (OISC). Am proiectat și simulat un procesor multi-core pe arhitectura SUBLEQ. Pentru testare am personalizat limbajul subleq cu multe funcționalități și am implementat un *assembler* care să compileze programe în fișiere binare executabile. Lucrarea conține detalii de implementare, noțiuni teoretice și referințe către alte lucrări asemănătoare.

Cuprins

1. INTRODUCERE

Aparatele de calcul au prins o perioadă rapidă de evoluție la începutul anilor 1940 odată cu apariția primului calculator care a folosit tuburi electronice cu vid. 10 ani mai târziu avem testul Turing, creat de matematicianul britanic Alan Turing și primul procesor comercial, apoi primul procesor care folosește tranzistori și primul limbaj de programare de nivel înalt, FORmula TRANslation (FORTRAN), dezvoltat la IBM. Totul s-a schimbat în anii 1970, când microprocesoarele intră în scenă. Până atunci toate procesoarele erau pentru uz comercial sau intern într-o companie, erau scumpe, aveau un consum ridicat dar erau foarte puternice pentru standardele de atunci. Microprocesoarele au revoluționat industria deschizând piața pentru aparate electronice de uz casnic și individual. Având mai puține instrucțiuni dar un cost mai mic de producție, erau perfecte și fezabile pentru a le integra în autovehicule, aparate casnice, console portabile de jocuri, etc. În prezent putem purta aceași discuție despre CISC vs RISC.

Limbajele de programare au evoluat și ele în timp ca să fie cât mai ușor de folosit. Dar există o categorie care s-a axat pe a face munca programatorului mai grea sau limbaje create pentru a experimenta un fel nou de programare. *“An esoteric programming language (ess-oh-terr-ick), or esolang, is a computer programming language designed to experiment with weird ideas, to be hard to program in, or as a joke, rather than for practical use.”*[18]. Aceste limbaje au ca scop cel puțin una dintre următoarele:

- *Minimalism*
- *New concepts*
- *Weirdness*
- *Themed*
- *Brevity*
- *Jokes*
- *Obfuscation*

La capitolul Minimalism putem vorbi de o întreagă specie de limbaje ezoterice, One Instruction Set Computer (OISC) [21]. Dar pe noi ne interesează **SUBstract and branch if Less than or EQual to 0 (SUBLEQ)** [23], din simplul motiv că e cel mai popular. Cum funcționează SUBLEQ și cât de popular este? Subleq e singura instructiune din limbaj, care poate fi considerat un limbaj de asamblare. Are 3 parametrii A B și C, toți sunt adrese. Procesorul va executa operația:

$$MEM[B] = MEM[B] - MEM[A]$$

și dacă rezultatul operației este mai mic sau egal cu 0 atunci procesorul sare cu execuția programului la adresa C.

SUBLEQ e atât de popular încât oamenii au început să scrie lucrări despre el, sa construiască sau să simuleze procesoare pe aceasta arhitectură, de asemenea și această lucrare e la fel. Există asambloare, există chiar mai multe versiuni ale limbajului, compilatoare și chiar un sistem de operare, DawnOS [6]. Următoarele paragrafele conțin câteva dintre motivațiile creatorului DawnOS [6], Geri. Sunt răspunsuri la întrebarea: "*Why technologies like Dawn and SUBLEQ are important?*".

"Your desktop computer have more than one billion of transistors in it, and it consumes 100 watts to read this text on it.",

"Because the detailed instruction manual of todays desktop (x86) CPU-s and IO system is approximately 50 000 page long, and no one completely understands how they work."

"Computers are driven by software written from 400 million lines of source code."

În capitolele ce urmează vom explora mai multe lucruri interesante despre OISC [21], **implementarea și simularea unui procesor multi-core cu arhitectura SUBLEQ [23]**, implementarea unui asamblor pentru a putea scrie cod ca să testăm procesorul, sintaxa limbajului și modul de utilizare a asamblorului, urmate de concluziile proiectului.

2. STUDIU BIBLIOGRAFIC

2.1 *A SIMPLE MULTI-PROCESSOR COMPUTER BASED ON SUBLEQ*

SUBLEQ nu e singurul limbaj sau arhitectură care face parte din categoria OISC dar e probabil cel mai popular și mai comun întâlnit în diferite lucrări. Alte variante asemănătoare includ *ADD and branch if Less than or EQal to 0 (ADDLEQ)*, care e la fel ca SUBLEQ dar folosește operația de adunare și *Plus 1 and brach if EQal (P1EQ)* care folosește operația de incrementare cu 1.

În lucrarea [8] autorii profită de simplitatea arhitecturii pentru a dezvolta în VHDL un procesor multi-core care, cel puțin în teorie poate să aibă un număr nelimitat de core-uri. Placa folosită, menționată în lucrare, este *Altera Cyclone III EP3C16*, o placă de cost redus și cu resurse limitate. Având în vedere capacitatea de memorie bloc valabilă pe placă, s-a estimat că e posibilă rularea a 28 de core-uri, fiecare cu o memorie cache de 2KB sau 56 de core-uri cu doar 1KB memorie cache individuală. Placa va fi programată și conectată la un calculator printr-un cablu USB după cum se vede în figura 2.1.

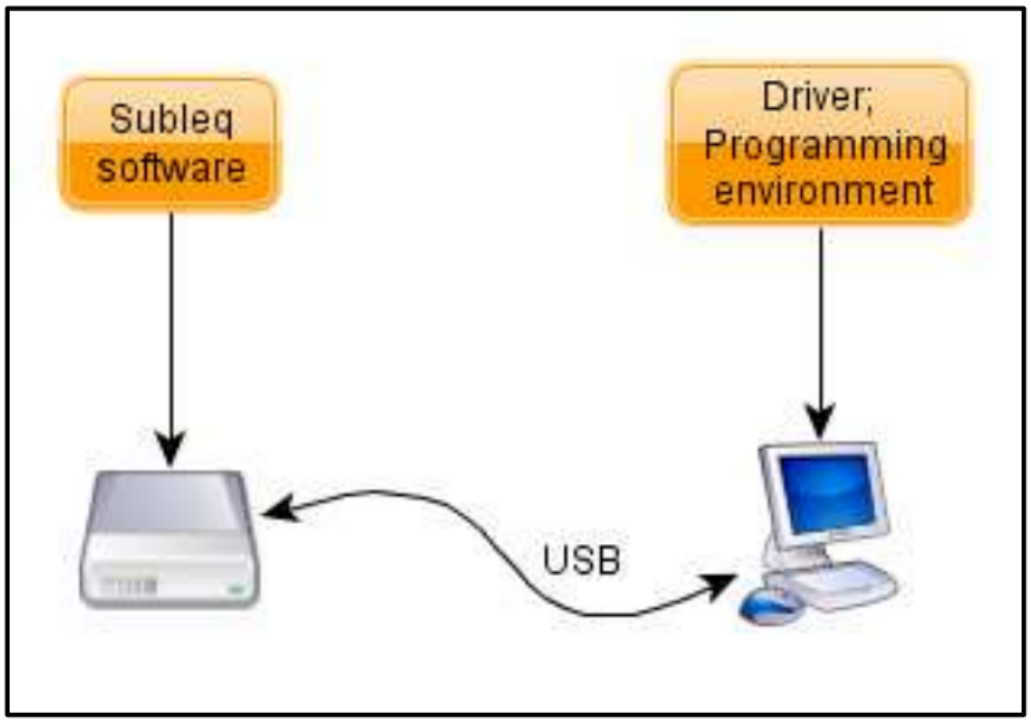

Figura 2.1: Diagramă a conexiunilor extrasă din [8]

Deoarece este foarte puțina memoria pe placă, este necesară o memorie externă care să acționeze drept RAM/ROM. Aici intervine calculatorul conectat la placă. Protocolul de comunicare SPI este folosit pentru a facilita o comunicare stabilă între cele 2 dispozitive, astfel *controller*-ul de memorie poate încărca instrucțiuni și date din memoria calculatorului în memoria cache construită pe placă, făcând-o accesibilă procesorului. Schema bloc este prezentată în figura 2.2.

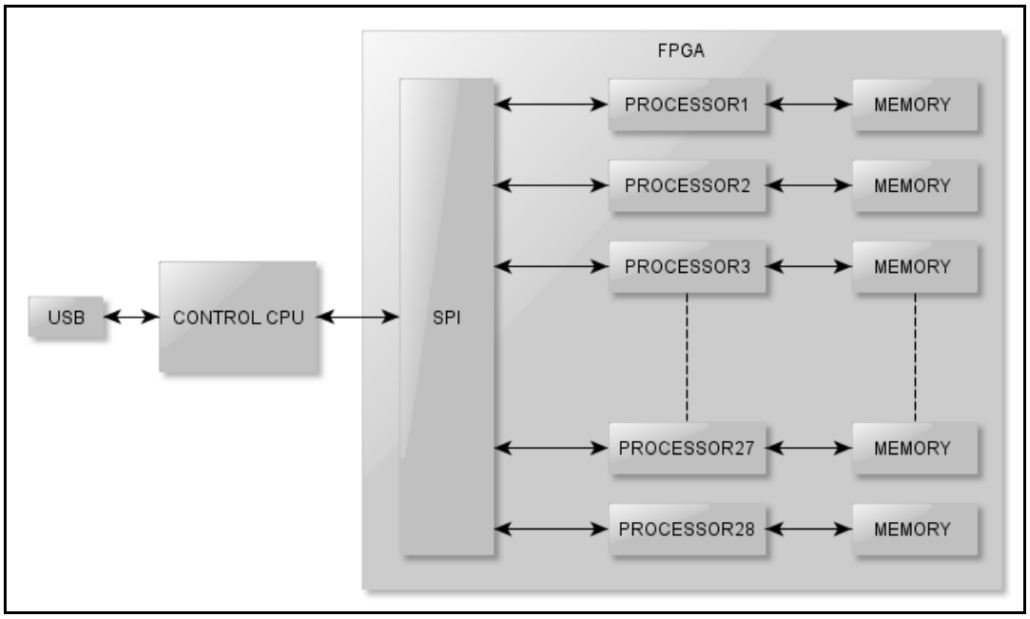

Figura 2.2: Diagramă bloc extrasă din [8]

Scopul final este acela de a compara performanța arhitecturii SUBLEQ cu altele care sunt folosite în industrie. Pentru a face acest lucru este necesară o metodă eficientă de a scrie cod care să fie compilat în instrucțiuni ce pot fi executate de procesor. Aceleași programe pot fi scrise ulterior într-un alt limbaj, precum C/C++, și rulate pe un dispozitiv cu procesor CISC. În capitolul 2 este prezentat pe scurt sintaxa limbajului de asamblare conceput de ei. În capitolul 4 este descris un limbaj simplu, cu sintaxă asemănătoare cu C, care va fi folosit pentru scrierea programelor. Fiecare subcapitol explică implementarea câte unui mecanism al limbajului, cât și codul aferent în limbaj de asamblare. Acestea sunt:

- *Stack*
- *Expressions*
- *Function calls*
- *Stack variables*
- *Multiplication*
- *Conditional jump*

Rezultatele arată că, deși are o performanță mult mai scăzută, un procesor OISC este fezabil, consumă puține resurse și are o viteză comparabilă cu procesoare mult mai complexe, atunci când numărul de core-uri e suficient de mare.

2.2 INKEL^{TM} PENTWICE MULTICORE PROCESSOR

Un procesor single-core e considerabil mai simplu decât un procesor multi-core. Chiar și efortul depus în construirea unui procesor cu doar 2 core-uri e mult mai mare. Acest lucru se datorează unui număr de factori, cum ar fi accesul la memorie și coerența memoriei cache. Totuși vine cu promisiunea că totul e scalabil până la un anumit număr foarte mare de core-uri unde apar alte probleme. Deci în teorie nu există diferență între 2, 32 sau 1000 de core-uri, din punct de vedere al efortului depus. În practică când ajungem la ceva atât de mare ca 1000 sau chiar și 100 lucrurile devin mai complicate.

În lucrarea [3] putem observa modificările care au fost făcute asupra procesorului *Inkel Pentium*, dezvoltat de aceeași echipă, pentru a-l transforma într-un procesor dual-core, numit *Inkel Pentwice*. Noul procesor va avea la bază 2 procesoare *Inkel Pentium* la care se vor aplica modificările necesare. Figura 2.3 arată cele 3 stagii prin care se trece pentru executarea fiecărei instrucțiuni: *Fetch→Decode→Either ALU or 5 stages of multiplication*

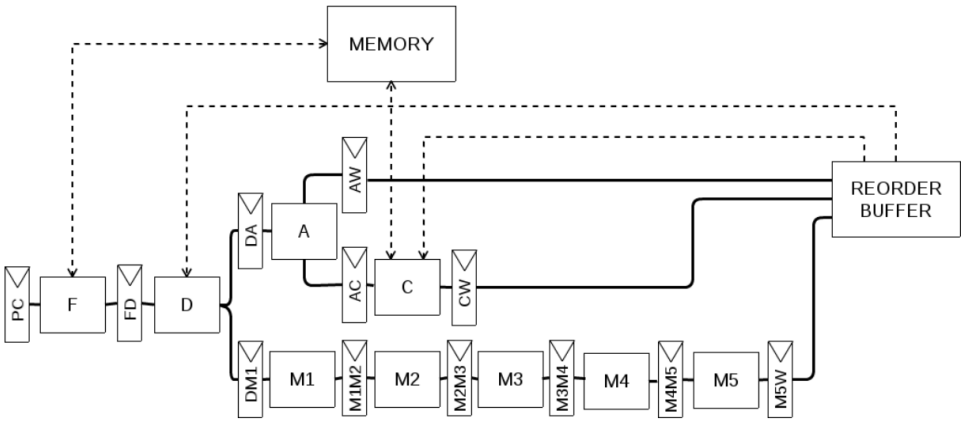

Figura 2.3: Pipeline general *Inkel Pentium* extrasă din [3]

Varianta originală avea un cache de nivel 1 între procesor si memorie. Acest lucru se modifică pentru noua variantă. Se adaugă un cache de nivel 2 comun pentru cele 2 core-uri și un circuit de arbitrare al accesului la bus. Aceste modificări asupra structurii memoriei se pot vedea în figura 2.4

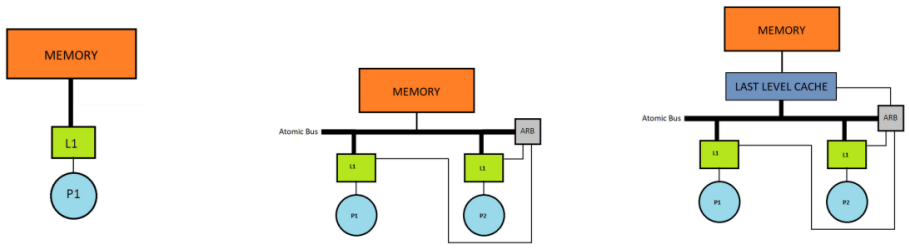

Figura 2.4: Diagrama bloc cu evoluția structurii memoriei extrasă din [3]

Coerența memoriei are legătură cu actualizarea datelor care sunt încărcate în cache-ul mai multor core-uri simultan. Considerând un scenariu simplu în care Core A și Core B

au același bloc de memorie în cache, iar Core A efectuează o operatie de scriere în acest bloc, în această situație blocul de memorie din cache-ul aferent lui Core B conține informații neactualizate, ca urmare apare un conflict ce poate duce la consecințe foarte grave. Pentru a rezolva problema coerenței trebuie implementat un protocol de coerență. În lucrare au ales să folosească cel mai simplu protocol, protocolul Valid-Invalid (VI), care se comportă după cum arată diagrama de stare din figura 2.5. Fiecare bloc de memorie din cache se poate afla în una din cele 2 stări. Revenind la exemplul dat, acum blocul din cache-ul aferent lui Core B trece în starea Invalid și va genera un *cache miss* la citire sau scriere.

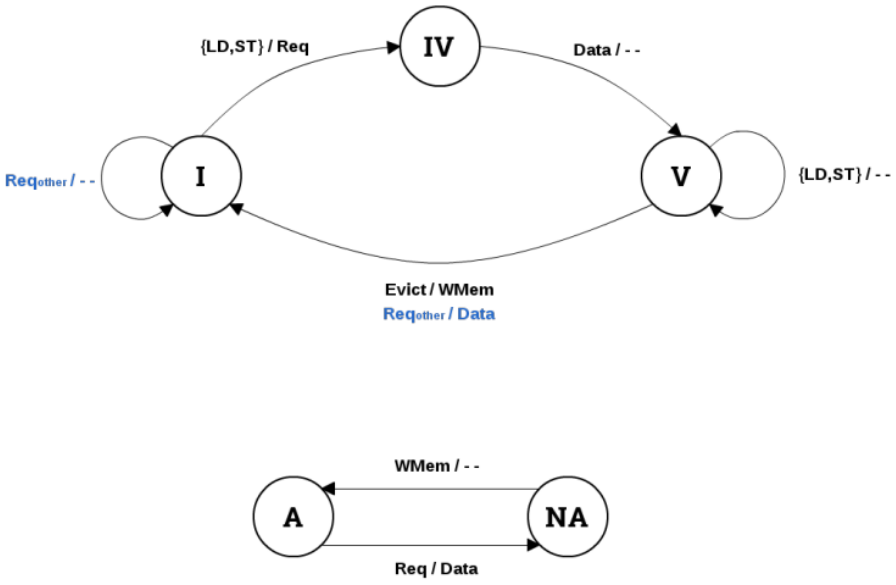

Figura 2.5: Protocolul VI extrasă din [3]

3. FUNDAMENTARE TEORETICA

3.1 SUBLEQ

3.1.1 Limbaje ezoterice

Limbajele de programare ezoterice se diferențiază față de cele obișnuite printr-o serie de factori, dar putem spune că eficiența sau ușurința de a le folosi nu se află printre factori. INTERCAL [20] este considerat primul limbaj ezoteric creat intenționat. În acest limbaj orice linie ce reprezintă o instrucțiune validă poate contine cuvântul cheie *"PLEASE"*. Compilatorul astfel se așteaptă la un anumit nivel de politețe din partea programatorului. Dacă programul are prea puține sau prea multe apariții ale cuvântului *"PLEASE"* atunci compilatorul va arunca o eroare. Aproximativ între $1/5$ până la $1/3$ din instrucțiuni trebuie să fie politicoase. Dacă o linie de cod nu este o instrucțiune validă atunci este ignorată. O metodă des întâlnită de a lăsa comentarii este de a folosi *"PLEASE NOTE"* urmat de textul comentariului. *"NOTE"* nu este o instrucțiune validă, motiv pentru care linia este ignorată, deci se comportă precum un comentariu.

Cele mai populare limbaje ezoterice sunt *Brainfuck* [15] și *Befunge* [13]. *Brainfuck* [15] a fost inventa de către Urban Müller în 1993 într-o încercare de a scrie cel mai mic compilator pentru sistemul de operare AmigaOS versiunea 2.0, rezultatul a fost un compilator de 240B. Limbajul consistă din o serie de comenzi ce servesc la manipularea unui vector de numere întregi, dimensiunea vectorului nu este precizată în specificațiile limbajului și deci implementări diferite lucrează cu vectori de dimensiuni diferite. *Befunge* [13] a fost inventat tot în 1993 de către Chris Pressey cu singurul scop de a fi cât mai greu de compilat. Este un limbaj bi-dimensional, ceea ce înseamnă ca folosește un grid cu o dimensiune fixă și ceea ce îl face greu de compilat este faptul că există comenzi ce modifică direcția de execuție a codului, astfel codul poate fi executat din toate cele 4 direcții cardinale. Ambele limbaje consistă numai din comenzi de o singură literă, ceea ce face codul foarte greu de citit și de înteles, dând un nou înteles cuvântului *obfuscat*.

Pe lângă cele menționate anterior există alte 2 limbaje care transcend timpul prin natura lor. Unul dintre ele e un limbaj despre care se știe doar că niște călători în timp au confirmat existența lui cândva în viitor, cât despre celălalt, are la bază doar conceptul de *loop* ce nu poate fi terminat decât prin legarea lui la durata de viață a unui număr sau concept pe care limbajul îl întelege. În teorie prin legarea unui *loop* la univers, acesta s-ar termina odată cu moartea universului. Folosind comanda *EXECUTE* limbajul poate fi folosit pentru a executa virtual orice, însă comanda poate fi folosită doar la finalul unui *loop*, ceea ce poate presupune așteptarea unei perioade lungi de timp.

Această secțiune servește drept introducere în minunata lume ezoterică a programării. Pentru mai multe detalii vizitați [18].

3.1.2 OISC

One Instruction Set Computer (OISC) sau *Ultimate Reduced Instruction Set Computer (URISC)* [21] este o categorie de limbaje de programare ezoterice care au o singură instrucțiune, deși nu toate limbajele din acestă categorie sunt *Turing complete* noi ne vom axa pe cele care sunt. Există 3 subcategorii de limbaje OISC [21]:

- *Transport Triggered Architecture*
- *Bit-manipulating machines*
- *Arithmetic-based Turing-complete machines*

Transport Triggered Architecture (TTA) e un design în care operațiile de calcul sunt efecte secundare a unui transport de memorie. De obicei niște regiștrii de memorie au o operație atribuită pe care o execută atunci când o operație de scriere are loc asupra registrului. De exemplu, într-un OISC care folosește o instrucțiune de copiere a unei valori dintr-o locație de memorie în alta, poate fi implementat prin declanșarea unor regiștrii ce conțin operații aritmetice și instrucțiuni de salt. Deși se utilizează o singură instrucțiune, este necesară implementarea *hardware* a operațiilor ce se vor a fi executate, listă care poate conține orice număr arbitrar de operații.

Bit-manipulating machines este cea mai simplă clasă, aceste limbaje se folosesc de operații logice pe biți sau manipularea memoriei la nivel de bit. Exemple:

- *FlipJump* [19] este probabil cel mai simplu OISC [21]. Arată cu succes că ai nevoie de aproape nimic pentru a putea face orice. Are o singură instrucțiune cu 2 parametrii A și B, adrese către un bit din memorie. Bitul care se află la adresa A este negat, apoi execuția sare necondiționat la adresa B, echivalent cu:

```
*A = !(*A);
JUMP B;
```

- *BitBitJump* [14] permite copierea unui bit dintr-o locație de memorie în altă locație de memorie, urmat de un salt necondiționat la o altă adresă. Se pare că acest proces este capabil de a executa orice operație, deoarece prin copierea biților se poate modifica codul ce urmează a fi executat. Instrucțiunea are 3 parametrii și se comportă astfel:

```
*A = *B;
JUMP C;
```

- *TOGA* [24] este un alt limbaj care are o singură instrucțiune cu 2 parametrii. Numele vine de la TOGgle And branch if the result is true. Particular pentru acest limbaj este separamea memoriei de date și memoriei de program. Din cauza acestui amănunt nu este *Turing complete*. Instrucțiunea se traduce în acest fel:

```
MEMD[A] = !MEMD[A];
IF (MEMD[A]) JUMP MEMP[B];
```

Arithmetic based Turing-complete Machines folosesc o operatie aritmetică și salt condiționat de rezultatul operației. În comparație cu celelalte 2 clase OISC, aceasta este *Turing complete* prin natura ei. Instrucțiunile operează cu numere întregi, care pot fi și adrese de memorie. Există o mulțime foarte mare de instrucțiuni în această subcategorie. Câteva exemple sunt:

- *ADDLEQ* [12] folosește operația de adunare:

```
*B = *B + *A;
IF (*B <= 0) JUMP C;
```

- *DJN* [17] folosește operația de decrementare:

```
*A = *A - 1;
IF (*A != 0) JUMP B;
```

- *P1EQ* [22] folosește operația de incrementare:

```
IF (*A + 1 = *B) JUMP C;
*B = *A + 1;
```

- *SUBLEQ* [23] folosește operația de scădere:

```
*B = *B - *A;
IF (*B <= 0) JUMP C;
```

- *Cryptoleq* [16] operează peste programe criptate:

```
*B = O1(*B, *A);
IF (O2(*B) <= 0) JUMP C;
```

  Operațiile O_1 și O_2 sunt definite în lucrarea [9] astfel:

$$O_1(x, y) = x^{-1} \cdot y \bmod N^2,$$

$$O_2(x) = \left\lfloor \frac{x-1}{N} \right\rfloor.$$

- *SUBLEQ+* este o versiune a lui SUBLEQ [23] care folosește numere negative pentru adresare indirectă a memoriei. Se presupune că această variantă este *Turing complete* și fără a folosi cod ce se modifică la execuție dar nu a fost demonstrat încă.

3.2 ASSEMBLER

3.2.1 *Two-Pass Assembler*

Un *assempler* este un program care traduce cod sursă, scris într-un limbaj de programare simbolic, în cod mașină, instrucțiune cu instrucțiune. Codul mașină poate fi după încărcat în memorie pentru a fi executat de către dispozitivul în cauză. Un *assembler* poate, și face mai mult de atât, ajută programatorul în mai multe aspecte ale scrierii programului.

Există mai multe tipuri de astfel de *traducătoare* dar în mare putem vorbi despre *one-pass assembler* și *two-pass assembler*. *One-pass assembler* citește fișierul sursă, care conține codul ce se dorește a fi tradus, o singură dată în timp ce *two-pass assembler* citește fișierul sursă de 2 ori și de fiecare dată face ceva diferit. În continuare ne interesează doar a doua categorie. Un *assembler* cu un singur pass are mai multe limitări care nu vor fi menționate aici dar le puteți găsi aici [11]. În figura 3.1 se poate vedea schema generală a unui *assembler*.

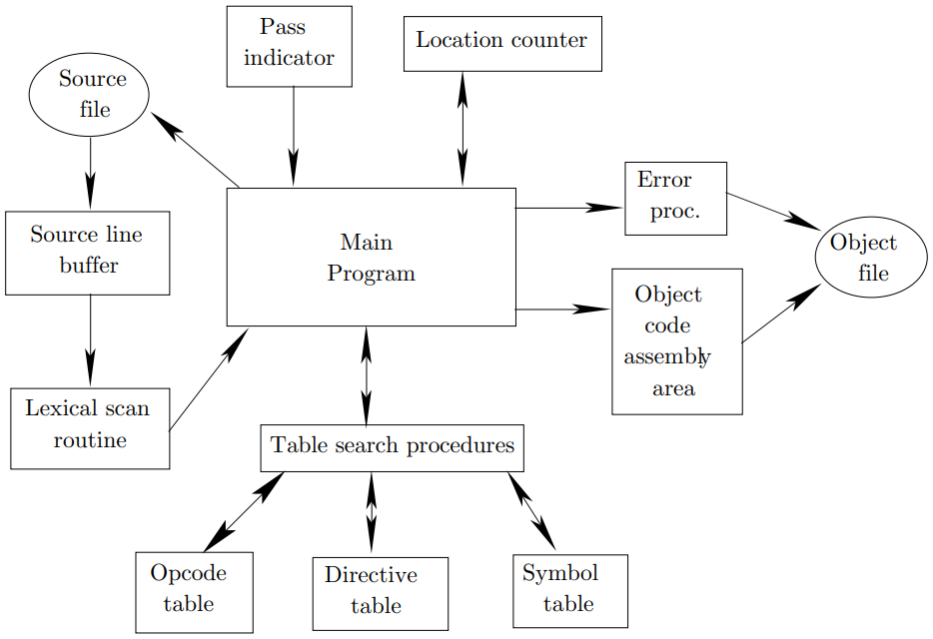

Figura 3.1: Componentele principale ale unui *assembler* extrasă din [11]

În primul pas citește fișierul sursă și caută toate *etichetele*. O etichetă este doar un nume format din caractere atribuit unei locații de memorie. Sunt foarte folositoare deoarece oamenii lucrează mai greu cu numere. O etichetă poate fi folosită pentru a da un nume unei variabile, constante, locație în cod unde se află o instrucțiune, funcții, MACRO și în funcție de limbaj pot fi și altele. În primul pas nicio instrucțiune nu este tradusă și toate etichetele sunt salvate într-un tabel pentru a fi folosite în pasul următor. Pentru a putea atribui unei etichete valoarea corectă, adică adresa corectă pe care o reprezintă, *assembler*-ul trebuie totuși să interpreteze fiecare instrucțiune. Acest lucru e necesar pentru că diferite instrucțiuni ocupă mai mulți *bytes* în memorie în funcție de numărul parametrilor dar și dimensiunea acestora.De asemenea unele directive se traduc în instrucțiuni sau date care sunt scrise în memorie. În figura 3.2 se vede ce se întâmplă în pasul 1. De cele mai multe ori, dar nu

e o regulă generală, la acest pas se construiește un fișier intermediar ce contine codul din fișierul sursă împreună cu informațiile culese și prelucrate în acest pas pentru a fi transmise la pasul 2.

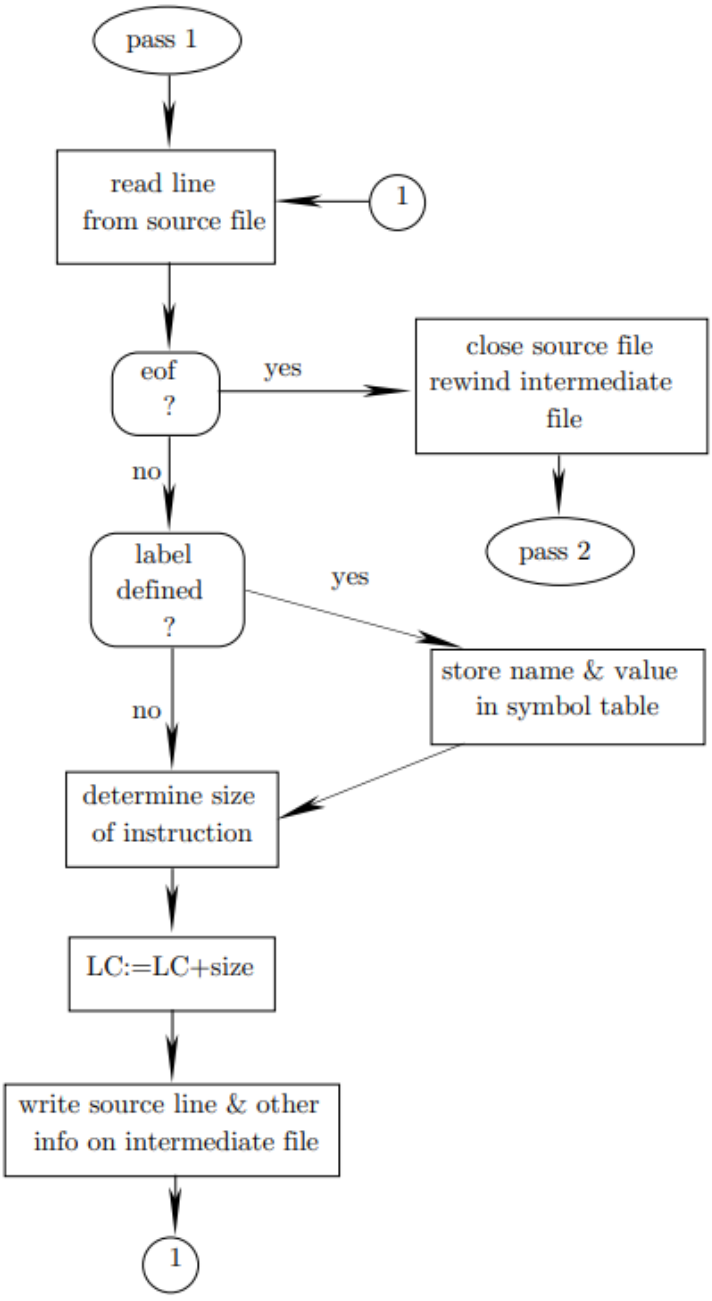

Figura 3.2: Primul pas extrasă din [7]

Pot să apară 3 probleme cu etichetele la pasul 1. Etichetele pot fi definite în mai multe locuri, numele unei etichete nu respectă regulile limbajului și etichete nedefinite dar utilizate. Dacă o etichetă este definită în mai multe locuri atunci cum se decide care este valoare ei? O etichetă ar trebui să fie un nume unic atribuit unei adrese pentru a putea fi ușor de folosit în cod. Multe limbaje nu permit simboluri speciale în numele etichetelor, de obicei sunt permise caracterele alfanumerice și bară orizontală '_' dar primul caracter nu poate fi o cifră. O etichetă este nedefintă dacă este folosită în cod fără a fi definită, deci nu

are o valoare atribuită.

Pasul al 2-lea se ocupă cu traducerea instrucțiunilor în cod mașină. La fel ca la pasul anterior citește fișierul intermediar și interpretează fiecare instrucțiune apoi o scrie în fișierul final. Când întâlnește o etichetă o înlocuiește cu valoarea ei. În figura 3.3 se vede ce se întâmplă în pasul 2.

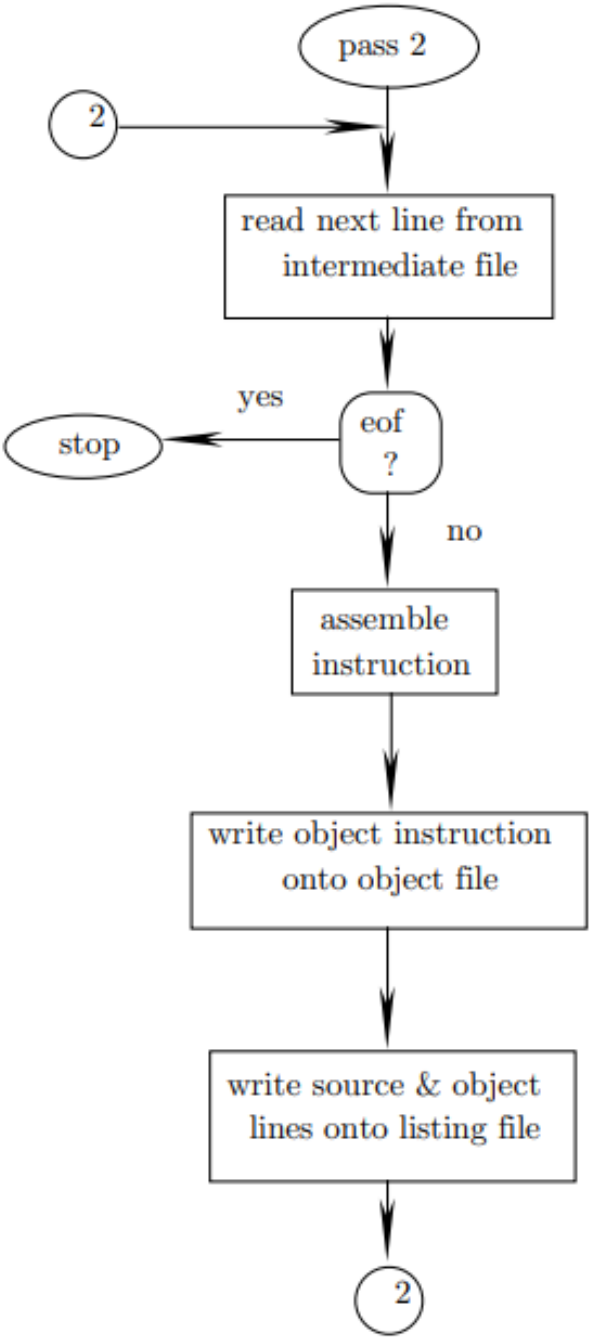

Figura 3.3: Al doilea pas extrasă din [11]

3.2.2 Directive

Aici începe frumusețea unui *assembler* dar nu aici se și termină. Pe lângă instrucțiuni un *assembler* oferă și directive, acestea sunt comenzi suplimentare care sunt executate sau interpretate de către *assembler* dar nu sunt traduse. Unele directive ar putea fi interpretate în instrucțiuni și apoi traduse dar directiva în sine nu e specifică limbajului. Pot fi împărțite pe categorii funcționale. Capitolul 3 din [11] este dedicat directivelor.

3.2.3 MACRO

În sine macro-urile sunt de asemenea directive dar pentru că sunt foarte folosite sunt considerate ca fiind ceva diferit și de sine stătător. Un macro e similar cu o subrutină dar există o diferență majoră între cele 2. O subrutină este o bucată de cod care e scrisă o singură dată și apoi poate fi folosită oriunde în cod prin simpla apelare. Un macro este tot o bucată de cod care e definit o singură dată și poate fi folosit de mai multe ori în cod dar de fiecare dată codul este copiat pur și simplu. Prin urmare subrutinele sunt tratate de către *hardware* la execuție, iar macro sunt tratate de *assembler* la compilare/asamblare/traducere. Pasul 0 este un pass special care e responsabil de a interpreta toate macro-urile și de a le expanda unde sunt folosite. Capitolul 4 din [11] este dedicat macro-urilor.

3.2.4 Analiză lexicală

În implementarea unui *assembler* analiza lexicală este primul și cel mai important proces. Un *assembler* lucrează cu fișiere text, fișierul sursă și cele intermediare. Un calculator lucrează cu codul mașină, el cunoaște doar 0 și 1 dar pentru oameni nu e deloc intuitiv, din cauza asta trebuie să lucrăm cu fișiere text care sunt usor de citit și de scris pentru noi. Analiza lexicală este primul proces care are loc în care codul sursă este transformat într-un șir de simboluri. Aceste simboluri sunt definite în funcție de nevoile limbajului, astfel încât să acopere toate elementele din cod. Exemplul următor explică aplicarea analizei lexicale asupra unei linii de cod dintr-un limbaj de programare arbitrar de nivel înalt:

```
int x = (a + b) * 2;

KEYWORD int
IDENTIFIER x
EQUALS
OPEN_PARANTHESIS
IDENTIFIER a
PLUS
IDENTIFIER b
CLOSE_PARANTHESIS
TIMES
LITERAL 2
SEMICOLON
```

Mai departe se trece la validarea sintaxei, o altă parte a analizei lexicale. Considerând exemplul anterior, trebuie să validăm că o paranteză deshisă este și închisă, dupa '=' trebuie să urmeze o expresie validă, ultimul simbol trebuie să fie `SEMICOLON`, după 'int' trebuie să urmeze un identificator, etc. În final simbolurile trebuie stocate și trimise către următorul proces împreună cu erorile găsite, după cum se vede în figura 3.4. Pentru explicații mai detaliate continuați cu [26].

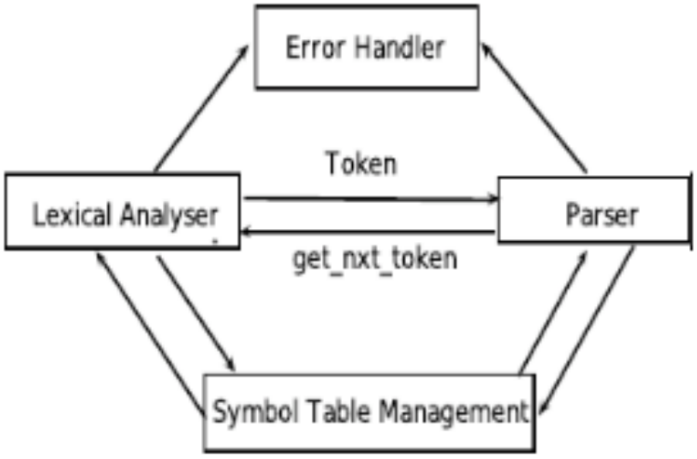

Figura 3.4: Analiză lexicală extrasă din [26]

3.3 PROCESOR

3.3.1 Automat cu stări finite

Procesorul este creierul unui calculator, cel care coordonează celelalte *organe* și ia decizii. Presupun că sursa de alimentare e inima atunci. Procesoarele care există azi sunt extrem de complexe, fie că sunt Complex Instruction Set Computer (CISC) sau Reduced Instruction Set Computer (RISC) tot au sute de instructiuni diferite [2]. Am obosit. Procesorul citește instrucțiuni din memorie, pe care le execută și enentual scrie înapoi în memorie. Deja avem un instrument foarte puternic dar nu e foarte util până nu introducem operații de intrare ieșire pentru a avea control asupra a ce se executa, cum și când.

Un automat cu stări finite e o mașina abstractă care se află în exact o stare din mulțimea stărilor în care se poate afla. Poate trece dintr-o stare în alta în mod necondționat sau condiționat de anumite date de intrare. Trecerea dintr-o stare în alta se numește tranziție. Și atunci un automat cu stări finite este definit de mulțimea stărilor, stare inițială în care se află, tranzițiile și ce date de intrare declanșează tranzițiile. Multe dispozitive automate nu au nevoie de circuite complexe pentru a opera. O ușă care se deschide automat poate fi implementată ca un automat cu stări finite cu 4 stări `(CLOSED, OPENING, OPEN, CLOSING)`, iar ca date de intrare are un semnal de un bit, 1 dacă senzorul de greutate e apăsat, 0 dacă nu e.

Procesoarele foarte simple pot fi implementate ca automate cu stări finite, precum în figura 3.5.

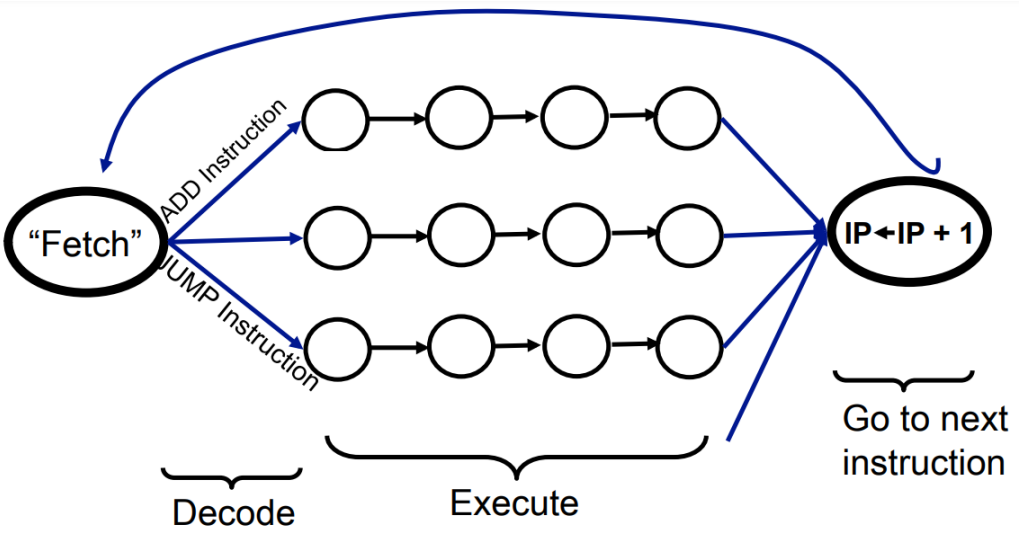

Figura 3.5: Procesor ca automat cu stări finite din [10]

3.3.2 Memoria cache

Prin memorie definim un dispozitiv de stocare a informației. Memoriile pot fi de foarte multe tipuri și pe multe categorii. Avem memorii volatile și nevolatile, după durata de viață a informației. Volatil inseamnă că în momentul în care sursa ce alimentează dispozitivul e oprită informația se șterge, e nevoie de un efort/energie pentru a menține starea memoriei. Memoriile nevolatile consumă energie la citire și scriere și păstrează permanent informația, cel puțin în teorie, din păcate totul e afectat de entropie. O altă categorie e dimensiunea spațiului de stocare și viteza de răspuns. Le luăm Impreună deoarece viteza de răspuns e în general invers proporțională cu dimensiunea spațiului de stocare. De asemenea viteza de răspuns crește cu cât distanța fizică dintre memorie și alt dispozitiv ce accesează memoria este mai mică. În figura 3.6 se poate observa ierarhia acestei categorii.

În figura 3.6 nu apare cea mai de jos categorie, care este *Harder Drive* [27]. Am vorbit despre limbaje de programare ezoterice [18], *Harder Drive* [27] e cam același lucru dar pentru spații de stocare a informației. Cele 3 exemple de implementări date în [27] sunt:

- ICMP ECHO sau PING în varianta cu *payload*
- Jocul TETRIS în care fiecare bloc ocupat înseamnă 1, altfel 0
- Teste COVID-19 electronice folosite, puse pe o placă PCB

Dacă ne întoarcem atenția către memoria cache, observăm că există mai multe nivele pe care e distribuită. Memoria cache de nivel 1 este cea mai aproape de procesor și în cazul unui procesor multi-core nu e împărțită, fiecare core are câte un cache L1. Nivelul 2 în acest caz este pentru perechi de core-uri, dar nu e neapărat să fie pentru fiecare 2, poate fi pentru fiecare 3 sau 4 sau mai multe core-uri, după caz. Nivelul 3 e cel mai mare, inclusiv la spațiu și la el au acces toate core-urile. Ierarhia aceasta există tocmai pentru a evita situația în care va trebui ca date sau instrucțiuni să fie încărcate din memoria RAM care e prea lentă

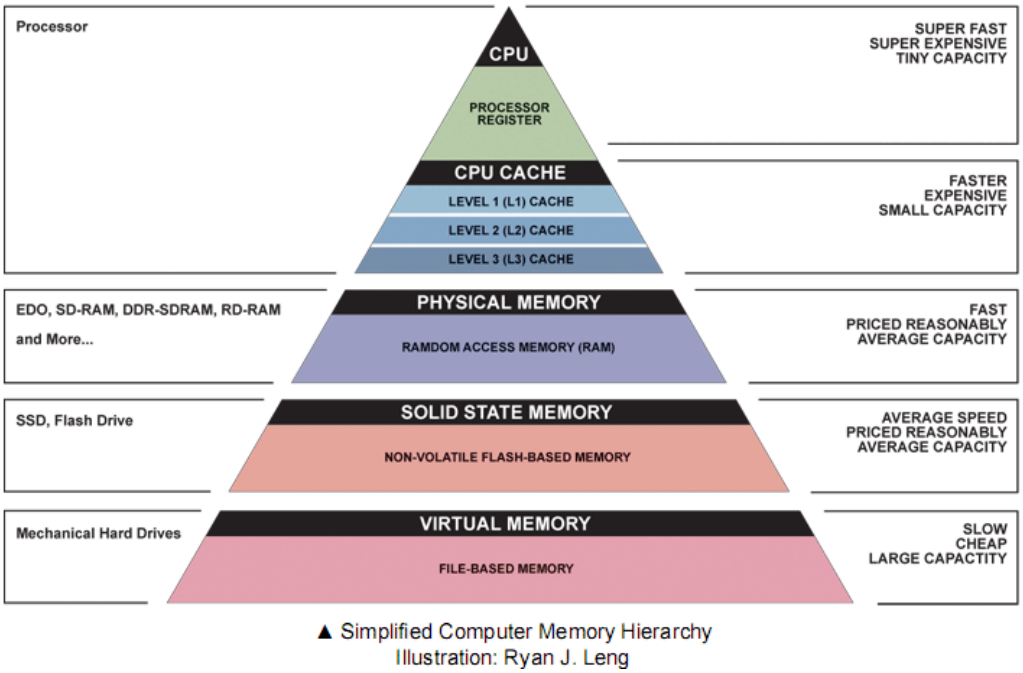

Figura 3.6: Ierarhia tipurilor de memorie din [25]

ca să țină pasul cu procesorul, după cum e descris în [5] unde sunt prezentate și alte soluții la această problemă.

Există mai multe tipuri de memorie cache. Se caracterizează prin comportamentul lor atunci când apare un cache miss la scriere sau la citire dar și după organizarea blocurilor de memorie. După organizarea blocurilor de memorie distingem cache cu mapare directă, total asciativă sau asociativă pe seturi. Maparea decide în ce linie din cache poate un bloc de la o anumită adresă să fie stocat. Politica de înlocuire decide ce linie va fi înlocuită în cazul unui *cache miss*, depinde de mapare, în cazul mapării directe un bloc de memorie poate fi pus într-o singură linie deci nu e nevoie de politică de înlocuire. Sunt 2 tipuri de strategii de scriere *write through*, informația e scrisă atât în cache cât și în memoria RAM și *write back*, informația e scrisă doar în cache, urmează să fie scrisă în memorie când linia e înlocuită. În cazul unui *cache miss* la scriere blocul poate fi încărcat în cache *write allocate* sau nu *no-write allocate* [7].

3.3.3 Sincronizarea circuitelor cu frecvențe diferite

Dacă nu ești tu inginerul care se ocupă de realizarea tuturor componentelor unui sistem atunci cel mai probabil nu toate au aceeași frecvență de operare. Chiar dacă ești, tot pot să existe motive pentru care asta se întâmplă, cum ar fi limitări fizice, reducerea costului sau reducerea consumului de energie, etc. Apare problema de comunicare sincronizată între 2 circuite. O soluție la această problemă este *hanshacking*, folosită de altfel și la rețele în protocolul TCP. Implementarea circuitului de sincronizare este redată în figura 3.7 și modul de funcționare în figura 3.8. Mai multe soluții găsiți aici [4]. Alte soluții precum *clock domain crossing* sunt amintite și aici [1].

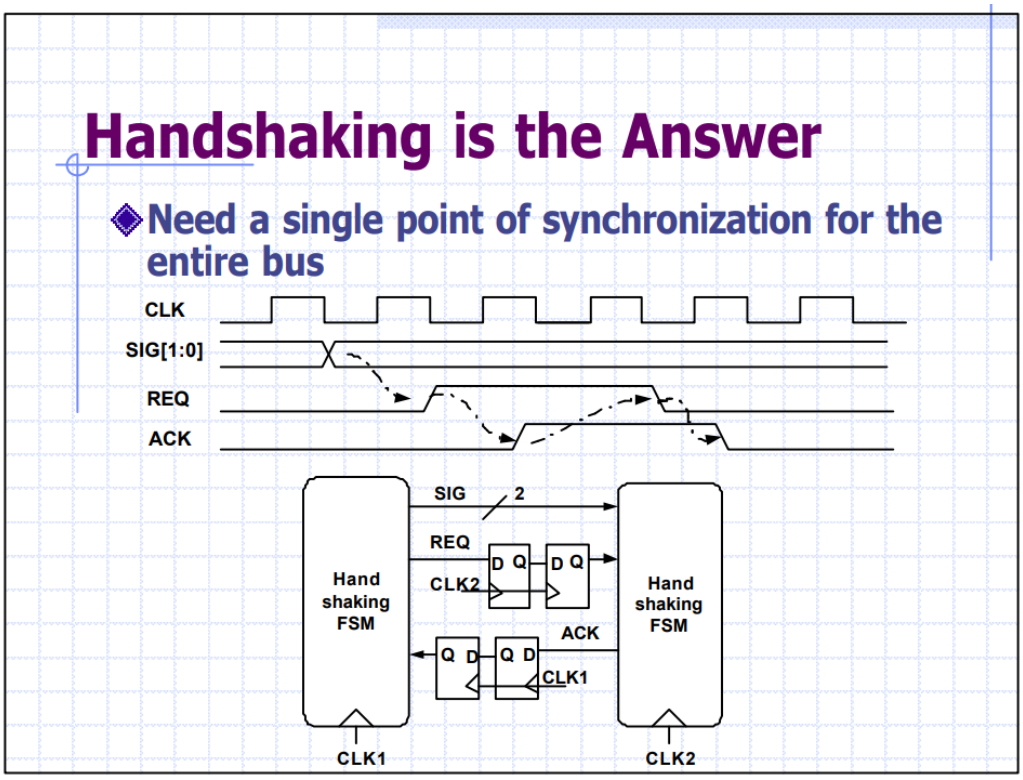

Figura 3.7: Sincronizare prin *handshaking* din [4]

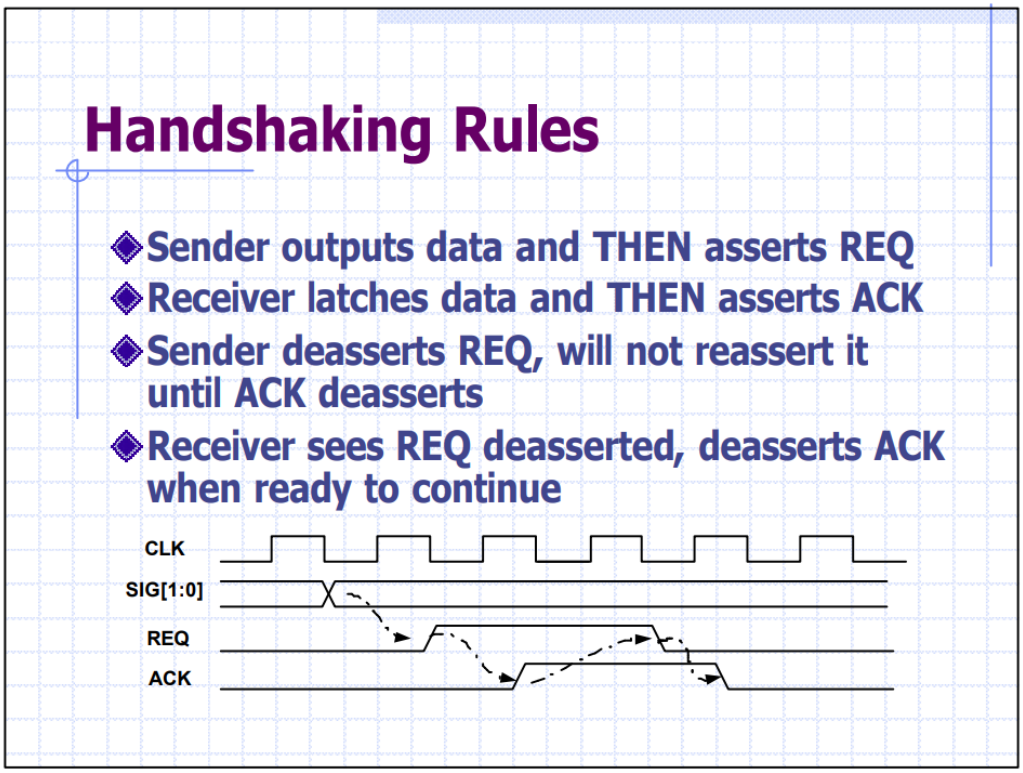

Figura 3.8: Sincronizare prin *handshaking* din [4]

4. IMPLEMENTARE

4.1 PROCESOR

4.1.1 Mediu de dezvoltare și generalități

Procesorul împreună cu celelalte circuite au fost proiectate în VHDL în mediul de dezvoltare *Active-HDL Student Edition*. *Active-HDL* pune la dispoziție instrumente pentru compilarea și simularea codului VHDL și Verilog. VHDL este un limbaj de programare hardware. Înseamnă că un program VHDL nu poate fi rulat pe un calculator, trebuie încărcat pe o placă FPGA, care pe baza codului va genera circuitele și conexiunile dintre ele. E o metodă rapidă de prototipare și testare pentru noi dispozitive hardware dar în același timp versiuni ieftine de plăci pot fi folosite în loc de proiectarea circuitelor electronice dedicate în anumite produse.

Proiectul conține următoarele circuite ce vor fi explicate în detaliu și care pot fi observate în figura 4.1. Pentru simplitate fiecare core al procesorului lucrează cu zone diferite de memorie pentru a elimina necesitatea de a implementa coerența memoriei cache, deoarece un core nu va modifica date care sunt accesibile de către celălalt core. Acest lucru a fost ușor de realizat, pentru procesor a fost adăugat un parametru generic ce reprezintă valoarea inițială a lui LC, Location Counter.

- Procesor cu arhitectură SUBLEQ
- Memorie cache cu mapare directă, *write-back* cu *write-allocate*
- Memorie RAM *single port*, citire asincronă și scriere sincronă
- Circuit de arbitrare cu prioritate
- Circuit de sincronizare print *handshaking*
- Bistabil de tip D folosit în implementarea circuitului de sincronizare (nu apare în figura 4.1)

O entitate e un noțiune VHDL care descrie semnalele unui circuit. Asemenea unei interfețe care poate avea mai multe implementări, o entitate poate avea mai multe arhitecturi. Aproape toate entitățile au și parametrii generici pentru a controla ușor numărul liniilor de date, numărul liniilor de adresă și dimensiunea memoriilor. Fiecare semnal are un prefix în nume pentru a specifica dacă este de intrare, de ieșire sau de intrare-ieșire.

4.1.2 Implementarea procesorului SUBLEQ

Arhitectura SUBLEQ a fost aleasă din cauza popularității ei în mediul online și din cauza simplității. Procesorul e implementat ca un automat cu stări finite care păstrează o stare internă cu informațiile necesare pentru a putea funcționa. În figura 4.2 se pot observa

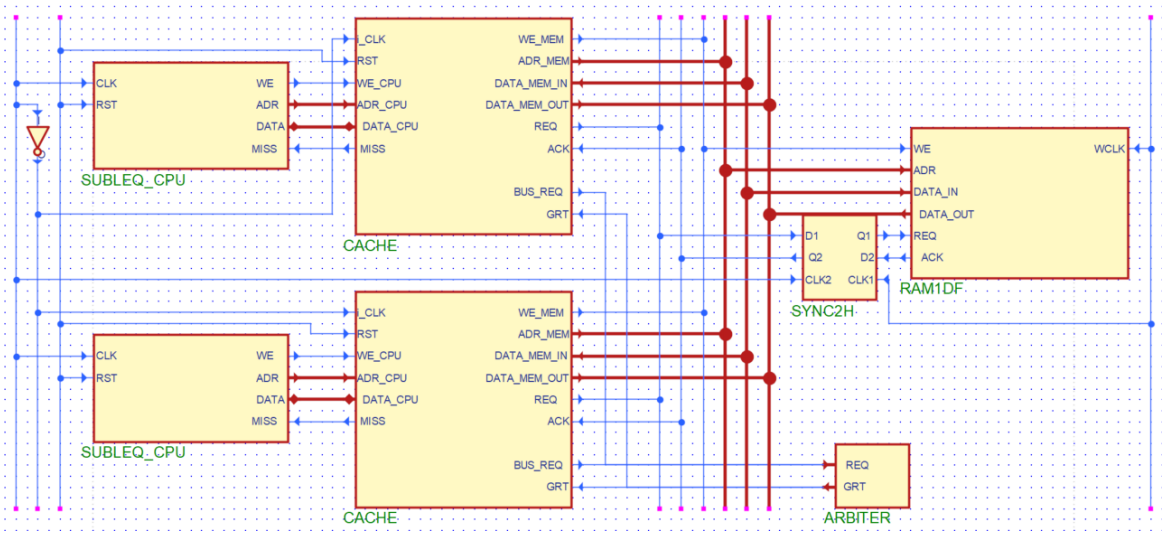

Figura 4.1: Schema bloc generală

stările și tranzițiile dintre acestea. Majoritatea stărilor și desigur, majoritatea instrucțiunilor din *pipeline* sunt cele de tip *fetch*. Executarea unei singure instrucțiuni SUBLEQ presupune 5 operații *fetch*, o operație ALU, o operație de scriere și încă una de decizie. Ca optimizare se verifică dacă noua valoare ce urmează să fie scrisă este diferită de cea care există deja la acea adresă, caz în care se sare peste operația de scriere, fiind redundantă. Optimizarea nu apare în figura 4.2.

```
entity SUBLEQ_CPU is
    generic(
        g_ADR_LINES  : integer := 64;
        -- bigger or equal to g_ADR_LINES due to SUBLEQ requirements
        g_DATA_LINES : integer := 64;
        g_LC_START   : integer := 0
    );
    port(
        i_CLK:   in    std_logic;
        i_RST:   in    std_logic;
        i_MISS:  in    std_logic;
        o_WE:    out   std_logic; -- WRITE ENABLE
        o_ADR:   out   std_logic_vector(g_ADR_LINES-1  downto 0);
        io_DATA: inout std_logic_vector(g_DATA_LINES-1 downto 0)
    );
end entity;
```

Listing 4.1: Entitatea procesorului

- i_CLK – semnal de intrare – semnalul de *clock*
- i_RST – semnal de intrare – semnal de reset, are același efect cu stare INIT din figura 4.2
- i_MISS – semnal de intrare – semnal de la cache, ‘1’ în cazul unui *cache miss* și ‘0’ pentru un *cache hit*

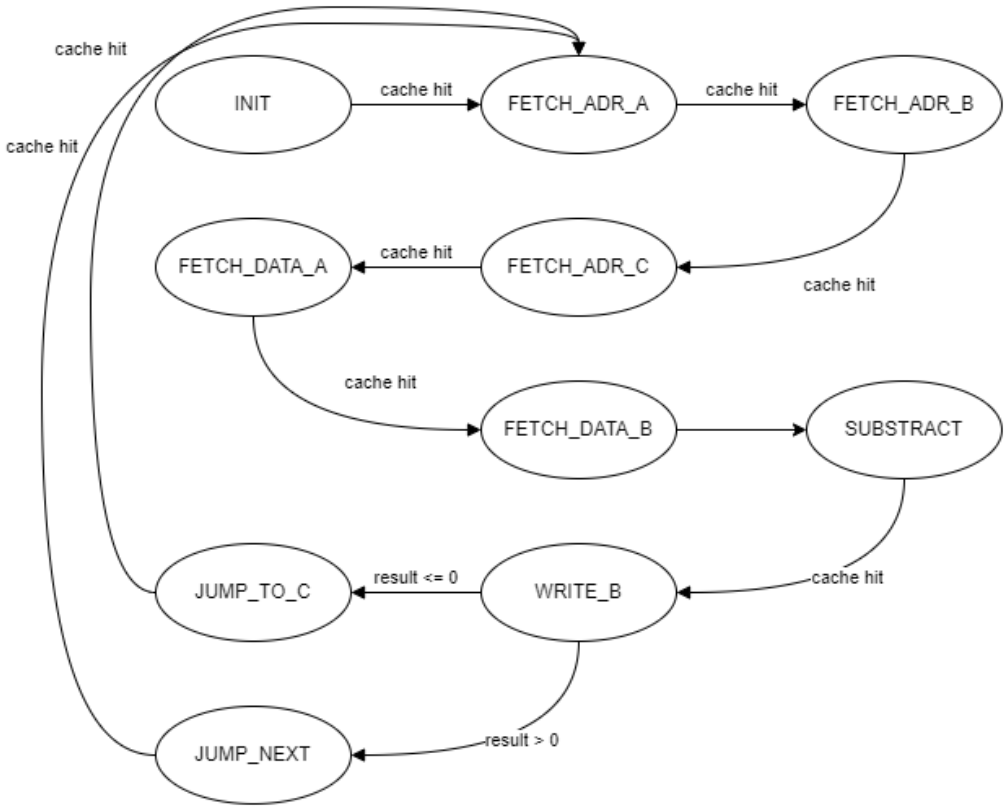

Figura 4.2: Diagrama de stare a procesorului

- o_WE – semnal de ieșire – ‘1’ pentru operația de scriere, ‘0’ pentru citire
- o_ADR – semnal de ieșire – adresa de la care se citește sau în care se scrie în cache
- io_DATA – semnal de intrare ieșire – contine datele ce vor fi scrise în cache sau citite din cache de la adresa specificată

Codul pseudocod pentru procesor:

```
LC = 0
WHILE( TRUE )
    A = MEM[LC]
    B = MEM[LC + 1]
    C = MEM[LC + 2]
    NEW_B = MEM[B] - MEM[A]
    IF( NEW_B != MEM[B] )
        MEM[B] = NEW_B
    IF( NEW_B <= 0 )
        LC = C
    ELSE
        LC = LC + 3;
```

4.1.3 Implementarea memoriei cache

Memoria cache e cea mai importantă componentă din întregul proiect, dat fiind numărul de operații *fetch* necesare doar pentru o singură instrucțiune, plus cea de scriere.

Pentru simplitate este folosită maparea directă. În acest caz avem un număr arbitrar de linii și o linie are un număr arbitrar de blocuri de dimensiunea unui *cuvânt*. Datele dintr-o anumita adresă din memoria de nivel superior, în cazul nostru memoria RAM, pot fi stocate într-o singură linie din cache după formula

$$LINE = ADR \bmod LINE\_COUNT$$

. Din cauza asta performanța este destul de mică și numărul de *cache miss*-uri este mai mare decât în cazul celorlaltor 2 tipuri de mapări.

Din cauză că fiecare instrucțiune poate presupune o operație de scriere în memorie, strategia de scriere aleasă a fost *write-back* cu *write-allocate*. Scrierea se face doar în cache, iar o linie din cache este scrisă în RAM doar când e înlocuită ca urmare a unui *cache miss*. Liniile conțin o informație suplimentară, un bit *Dirty Flag* care specifică dacă o linie trebuie sau nu să fie scrisă în memoria RAM. *Dirty Flag* e setat pe 1 când are loc o operație de scriere în cache, în felul ăsta nu se face scrierea liniei dacă nu e necesar. Figura 4.3 conține diagrama de stare a memoriei cache.

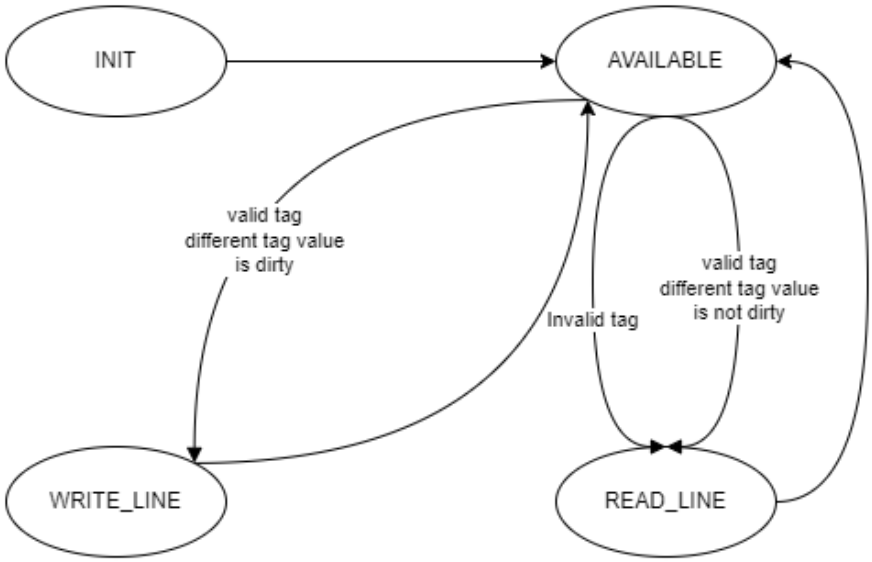

Figura 4.3: Diagrama de stare a memoriei cache

Distingem 4 situații din figura 4.3:

- Linia din care vrem să citim sau să scriem are tagul invalid, ceea ce înseamnă că încă nu a fost încarcată din memoria RAM nicio informație utilă, linia respectivă e goală sau contine informație inutilă. Se citește linia din memoria RAM și se răspunde cu *cache miss*.
- Tagul extras din adresa de la care vrem să citim sau să scriem e diferit de tagul liniei și are *Dirty Flag* setat pe 0, nu s-a scris în acea linie. Linia se înlocuiește și se răspunde cu *cache miss*.
- Tagul extras din adresa de la care vrem să citim sau să scriem e diferit de tagul liniei

și are *Dirty Flag* setat pe 1, s-a scris în acea linie. Prima dată linia se este scrisă în memoria RAM și apoi se înlocuiește. Se răspunde cu *cache miss*.

- La citire sau scriere dacă tagul liniei e valid și tagul extras din adresă e același cu tagul liniei, se execută operația cerută și se răspunde cu *cache hit*.

```
1  entity CACHE is
2      generic(
3          g_ADR_LINES  : integer := 64;
4          g_DATA_LINES : integer := 64;
5          g_LINE_SIZE  : integer := 8;
6          g_MEM_LINES  : integer := 32
7      );
8      port(
9          i_CLK:      in    std_logic;
10         i_ADR:      in    std_logic_vector(g_ADR_LINES-1  downto 0);
11         i_DATA_MEM: in    std_logic_vector(g_DATA_LINES-1 downto 0);
12         i_WE:       in    std_logic;
13         i_ACK:      in    std_logic;
14         i_RST:      in    std_logic;
15         i_GRT:      in    std_logic;
16         o_ADR:      out   std_logic_vector(g_ADR_LINES-1  downto 0);
17         o_DATA_MEM: out   std_logic_vector(g_DATA_LINES-1 downto 0);
18         o_WE:       out   std_logic;
19         o_MISS:     out   std_logic;
20         o_REQ:      out   std_logic;
21         o_AR_REQ:   out   std_logic;
22         io_DATA:    inout std_logic_vector(g_DATA_LINES-1 downto 0)
23     );
24 end entity;
25
```

Listing 4.2: Entitatea memoriei cache

- i_CLK – semnal de intrare – semnalul de *clock*
- i_ADR – semnal de intrare – adresa cerută de procesor pentru a citi din cache sau scrie în cache.
- i_DATA_MEM – semnal de intrare – datele returnate de memoria RAM pentru adresa cerută
- i_WE – semnal de intrare – semnal de la procesor, ‘1’ pentru scriere, ‘0’ pentru citire
- i_ACK – semnal de intrare – semnal de *acknewledge* de la memoria RAM pentru sincronizare
- i_RST – semnal de intrare – semnal de reset, are același efect cu stare INIT din figura 4.3
- i_GRT – semnal de intrare – semnal de la arbitru, ‘1’ înseamnă acces la *bus* pentru a comunica cu memoria RAM

- o_ADR – semnal de ieșire – adresa de la care se citește sau în care se scrie în RAM
- o_DATA_MEM – semnal de ieșire – semnal către RAM cu datele ce se vor scrie la adresa specificată
- o_WE – semnal de ieșire – semnal către RAM, ‘1’ pentru scriere, ‘0’ pentru citire
- o_MISS – semnal de ieșire – semnal către procesor, ‘1’ în cazul unui *cache miss*, ‘0’ pentru *cache hit*
- o_REQ – semnal de ieșire – semnal de *request* pentru scriere sincronă în RAM
- o_AR_REQ – semnal de ieșire – semnal către arbitru pentru a cere acces la *bus*
- io_DATA – semnal de intrare ieșire – pentru procesor, contine datele ce vor fi scrise în cache sau citite din cache de la adresa specificată

4.1.4 Implementarea celorlaltor circuite

Memoria RAM

Memoria RAM servește drept spațiu de stocare principal. Datele sunt încărcate dintr-un fișier binar la începerea simulării. Dimensiunea e specificată printr-un parametru generic. Are un singur port, ceea ce înseamnă că doar o singură operație, de citire sau scriere, poate fi executată la un moment dat. Citirea e asincronă și de dragul simplității presupunem că are loc destul de repede pentru a fi citită într-un singur ciclu de tact. Scrierea e sincronă, se execută doar la semnal de *clock* pe front crescător.

```
1   entity RAM1DF is
2       generic(
3           g_ADR_LINES  : integer := 64;
4           g_DATA_LINES : integer := 64;
5           g_MEM_SIZE   : integer := 256
6       );
7       port(
8           i_DATA: in  std_logic_vector(g_DATA_LINES-1 downto 0);
9           i_ADR:  in  std_logic_vector(g_ADR_LINES-1  downto 0);
10          i_WE:   in  std_logic;
11          i_WCLK: in  std_logic;
12          i_REQ:  in  std_logic;
13          o_DATA: out std_logic_vector(g_DATA_LINES-1 downto 0);
14          o_ACK:  out std_logic := '0'
15      );
16  end entity;
```

Listing 4.3: Entitatea memoriei RAM

- i_DATA – semnal de intrare – conține datele ce se vor scrie la adresa specificată
- i_ADR – semnal de intrare – adresa din care se citește sau în care se scrie
- i_WE – semnal de intrare – ‘1’ pentru scriere, ‘0’ pentru citire

- i_WCLK – semnal de intrare – semnal de *clock* pentru scrierea sincronă
- i_REQ – semnal de intrare – semnal de *request* pentru scriere
- o_DATA – semnal de ieșire – returnează datele care se află la adresa cerută
- o_ACK – semnal de ieșire – semnal de *acknowledge* trimis după ce s-a executat scrierea, folosit pentru sincronizare

Circuit de arbitrare

Un singur core poate avea acces la *bus* la un moment dat. Circuitul de arbitrare se asigură de asta. Funcționează pe bază de prioritate, înseamnă că primul core va avea tot timpul acces la *bus* în timp ce al doilea are acces doar dacă primul core nu trimite cerere. Este implementat pentru a fi scalabil. Un parametru generic desemnează câte intrări și ieșiri are circuitul. În 4.1 are 2 intrări și ieșiri pentru cele 2 core-uri.

```
1  entity ARBITER is
2      generic(
3          g_SIZE : integer := 2
4      );
5      port(
6          i_REQ: in  std_logic_vector(g_SIZE-1 downto 0);
7          o_GRT: out std_logic_vector(g_SIZE-1 downto 0)
8      );
9  end entity;
```

Listing 4.4: Entitatea arbitrului

- i_REQ – semnal de intrare – semnale de la alte circuite pentru a cere acces la *bus*
- o_GRT – semnal de ieșire – oferă acces la *bus* unui singur circuit

Circuit de sincronizare

Este folosit pentru a asigura că în timpul unei operații de scriere, semnalele implicate nu își schimbă valoarea până când se primește un semnal de la memoria RAM cum că scrierea a fost finalizată. Cu alte cuvinte circuitul asigură sincronizarea comunicării între 2 circuite care operează la diferite frecvențe.

```
1  entity SYNC2H is
2      port(
3          i_D1:   in  std_logic;
4          i_D2:   in  std_logic;
5          i_CLK1: in  std_logic;
6          i_CLK2: in  std_logic;
7          o_Q1:   out std_logic;
8          o_Q2:   out std_logic
9      );
10 end entity;
```

Listing 4.5: Entitatea circuitului de sincronizare

- i_D1 – semnal de intrare – semnal de *request*
- i_D2 – semnal de intrare – semnal de *acknowledge*
- i_CLK1 – semnal de intrare – semnal de *clock* a circuitului care face *request*
- i_CLK2 – semnal de intrare – semnal de *clock* a circuitului care face *acknowledge*
- o_Q1 – semnal de ieșire – semnal de *request* întârziat în funcție de CLK2
- o_Q2 – semnal de ieșire – semnal de *acknowledge* întârziat în funcție de CLK1

Bistabil tip D

Circuitul de sincronizare folosește câte 2 bistabile tip D pentru fiecare semnal pentru a le întârzia cu câte 2 cicluri de tact.

```
1    entity D_FF is
2        port(
3            i_D:   in  std_logic;
4            i_CLK: in  std_logic;
5            o_Q:   out std_logic := '0'
6        );
7    end entity;
```

Listing 4.6: Entitatea bistabilului tip D

- i_D – semnal de intrare – conține datele ce vor fi salvate la următorul front crescător al semnalului de *clock*
- i_CLK – semnal de intrare – semnal de *clock*
- o_Q – semnal de ieșire – returnează datele curent salvate

4.1.5 Rezultatul simulării

Circuitul din figura 4.1 a fost simulat în mediul de dezvoltare *Active-HDL Student Edition* În figura 4.4 este prezentă diagrama de timp cu semnalele, în care s-a considerat un exemplu mai simplu cu un singur core pentru a fi mai ușor de urmărit și de explicat. Printre alte considerente, dimensiunea unui cuvânt este 8 biti sau 1 octet, iar memoria cache are o singură linie cu 8 blocuri pentru a fi siguri că va avea loc o scriere în memoria RAM. Explicația pe segmente:

- Segmentul 0: Conține doar partea de inițializare a circuitelor, fiecare având nevoie de un ciclu de tact.
- Segmentul 1: Procesorul face prima operație de citire la adresa 0x00, la acest moment singura linie din cache e neinițializată, adică are tagul invalid. Memoria cache trece în starea READ_LINE și execută 8 citiri din memoria RAM, deoarece linia are 8 blocuri de memorie de dimendiunea unui cuvânt. În tot acest timp se trimite un semnal de *cache miss* către procesor, iar el știe să aștepte până la un semnal *cache hit* așa cum apare și în figura 4.2.

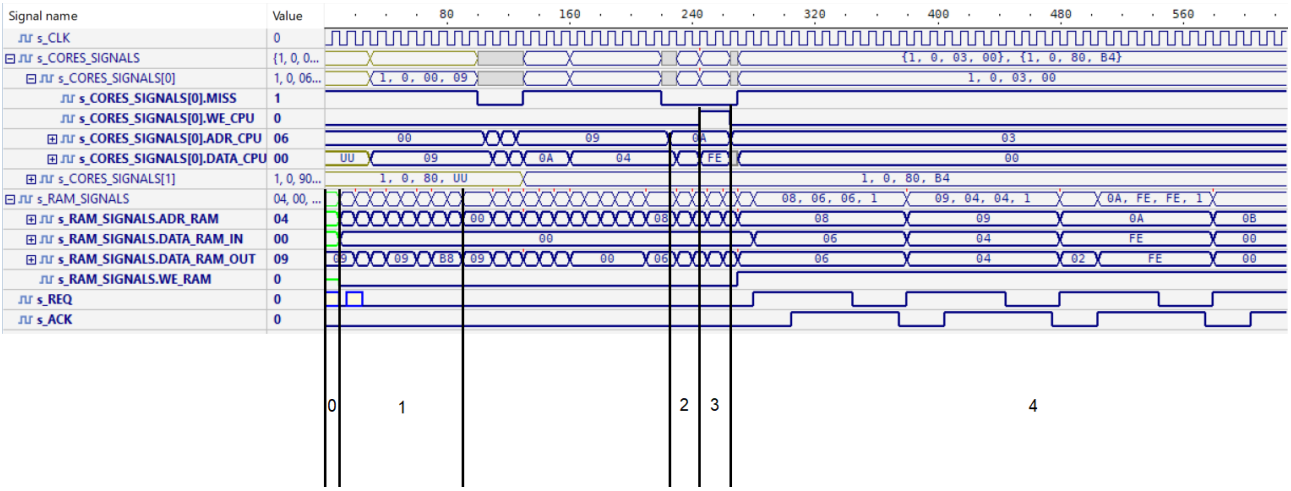

Figura 4.4: Diagrama de timp pe exemplu simplu

- Segmentul 2: Aici are loc ultima operație *fetch*, urmată de operația de scădere.
- Segmentul 3: Se execută operația de scriere prin setarea semnalului WE_CPU pe '1'. Se scrie în aceeași locație de unde s-a facut ultima citire, motiv pentru care adresa rămâne aceeași.
- Segmentul 4: Procesorul cere datele de la adresa 0x03. Deoarece există doar o linie de cache cu 8 blocuri, asta înseamnă că adresa 0x03 se află în linia ce conține adresele 0x00 - 0x07. Linia curentă e 0x08 - 0xFF, știm asta pentru că ultima operație s-a executat pe adresa 0x0A. Se trimite un semnal *cache miss* și pentru că s-a scris în linie trebuie să fie scrisă în RAM înainte de a fi înlocuită. Se observă 3 operații complete de scriere și una incompletă împreună cu semnalele de sincronizare aferente. În total o singură scriere durează 10 cicluri de tact. În cazul primelor 2 scrieri, informația scrisă este aceiași cu cea care exista deja la adresele respective. La a 3-a scriere, în RAM este valoarea 0x02 și valoarea scrisă este 0xFE (-2 în complement de 2).

4.2 ASSEMBLER

4.2.1 Considerente generale

Proiectul a fost dezvoltat în C, mai exact standard ISO C14. Alegerea nu a fost arbitrară, C este un limbaj de programare foarte puternic și capabil. O alternativă luată în considerare a fost C# dar stilul de programare OOP lasă de dorit și tentația de a complica lucrurile folosind funcționalități de limbaj există. Tipul de *assempler* ales este *two-pass assembler*. Acest tip este mai ușor de înteles și e mai extensibil decât un *one-pass assembler*. Asta înseamnă că execută 2 pași asupra fișierului cu codul sursă. Cei 2 pași au responsabilități diferite.

Primul pas se ocupă de strângerea de informații necesare pentru pasul 2 și de transmiterea acestor informații. În mod normal se face printr-un fișier intermediar, în cazul nostru fișierul intermediar nu este unul standard. Acesta conține o versiune mai simplă a fișierului sursă în loc să conțină una mai complicată. Etichetele și macro-urile găsite în pasul 1 se

păstrează în memorie, de unde pasul 2 le accesează. Fișierul intermediar conține o versiune mai curată a fișierului sursă și macro-urile expandate. Din cauza aceasta opțiunea de listare a codului final într-un fișier nu este posibilă, la pasul 2 nu se știe cum arată fișierul sursă. Toate erorile sunt prinse la pasul 1. Pasul 2 traduce efectiv instrucțiunile în cod mașină, executabil de către procesor.

Pentru a face lucrurile mai simple, o linie poate avea maxim 256 de caractere. Tot ce se află după este ignorat. Simbolul `ZERO` este singurul care este scris de *assembler*, este o etichetă către o zonă de memorie ce mereu va avea valoarea 0. Aceasta regulă este consolidată în cod și o încercare de a modifica valoarea va rezulta într-o eroare de compilare. Este posibil ca din cod să se facă referire la valoarea lui LC folosind simbolul '*'. Pentru a face referire la adresa următoarei instrucțiune se folosește simbolul '?'. Atunci instrucțiunea `NOP`, o instrucțiune care nu face nimic, se scrie `subleq ZERO ZERO ?`.

4.2.2 Aspecte importante din cod

Niciodată nu se alocă memorie în mod dinamic. Informațiile referitoare la etichete și macro-uri sunt salvate într-un vector alocat static. Referința la o etichetă sau macro se face prin indexul acestora în vector.

```
1   typedef int16_t static_ptr;
2   ...
3   typedef struct line_def {
4     static_ptr label;
5     static_ptr macro;
6     LINE_TYPE type;
7     DIRECTIVE_TYPE directive;
8     PARAM_DEF params[5];
9   } LINE_DEF;
10  ...
11  LABEL_DEF labels[INT16_MAX];
12  int labels_count;
13  MACRO_DEF macros[INT16_MAX];
14  int macros_count;
```

Listing 4.7: Referință prin index în structura LINE_DEF

În interiorul unui macro ar trebui să fie posibil să definești etichete. Problema este că un macro este făcut pentru a fi folosit în mai multe locuri. Asta ar duce la definirea multiplă a unei etichete, situație care rezultă în eroare de compilare. Pentru a rezolva problema, etichetele au acum un domeniu de definiție, care poate fi global sau macro. O etichetă cu domeniu macro nu are valoare. Când macro-ul este expandat se crează o etichetă globală cu valoarea curentă a lui LC, la numele căreia se adaugă un sufix format din 4 litere și cifre alese la întamplare. În felul acesta o etichetă poate fi definită într-un macro.

```
1    int macro_expand(int lc, MACRO_DEF macro, PARAM_DEF* params, FILE* stream) {
2        ...
3        for (int i = macro.first_line; i < macro.first_line + macro.line_count;
     i++) {
4            LINE_DEF line = lines[i];
5
6            if (line.label != -1) { // line contains a label
7              label = labels[line.label];
8              random_padding_after(label.name, 4, seed);
9              label.value = lc;
10             label.scope = LABEL_SCOPE_GLOBAL;
11             label_add(label);
12           }
13       ...
14       }
15   }
```

Listing 4.8: Tratarea unei etichete în interiorul unui macro

Un macro poate fi folosit în interiorul definiției unui alt macro. Pentru a nu permite crearea unui ciclu infinit de expandare a fost luată decizia de a nu permite folosirea unui macro înainte de a fi definit. Pentru a permite folosirea unui macro înainte de a fi definit se poate testa că nu există cicluri prin parcurgerea grafului orientat al dependințelor dintre macro-uri. Dacă un nod al grafului este vizitat de 2 ori atunci există un ciclu și trebuie semnalată o eroare.

4.2.3 Sintaxă limbaj

Instrucțiune

Singura instrucțiune este *subleq*, din acest motiv este implicită și se scriu doar parametrii. O instrucțiune poate fi scrisă cu 3 parametrii, 2 parametrii sau 1 parametru. În cazul cu 3 parametrii se execută instrucțiunea subleq obișnuită `subleq A B C`. În cazul cu 2 parametrii, al 3-lea parametru se pune automat '?', `subleq A B ?`. În felul acesta tot timpul se trece la instrucțiunea următoare. În cazul cu un singur parametru se întelege `subleq A A ?` care e echivalent cu `A = A - A` sau `A = 0`. S-a luat în considerare ca forma cu un parametru să se traducă drept `ZERO ZERO A`, adică salt necondiționat dar are mai puțină utilitate decât setarea pe 0. Incrementează LC cu 3.

Etichetă

Orice linie poate avea o etichetă. Trebuie să apară prima pe linie, se delimitează de restul liniei prin simbolul ':', de exemplu `init: a a ?`.

Directive

Pe baza lucrării [11] a fost luată decizia ca directivele să înceapă cu caracterul '.'. Directivele prezente în limbaj sunt:

- `.ORG <number>`
  Are un singur parametru numeric. Setează valoarea lui LC. Este interpretată în pasul 2 prin scrierea de valori 0x00 până se ajunge la noua valoare a lui LC.

- `.DATA <number>/<A>`
  Are un singur parametru care poate fi un număr sau o etichetă. Stochează în memorie valoarea precizată și incrementează LC cu 1. Este interpretată în pasul 2 prin scrierea în memorie a valorii numerice sau valorii etichetei.

- `.END`
  Nu are parametrii. Crează un ciclu infinit. Este interpretată în pasul 1 prin expandarea în instrucțiunea `ZERO ZERO *`.

- `<label> .MACRO <p1> <p2> <p3> <p4> <p5>`
  Trebuie să aibă o etichetă ce va acționa drept nume al macro-ului. Poate avea între 0-5 argumente, toate sunt etichete. Nu incrementează LC și nici liniile următoare nu îl incrementează. După această directivă pot să urmeze oricâte linii care conțin instrucțiuni, alte directive sau macro-uri. Aceste linii formează corpul macro-ului. Este interpretată în pasul 1 prin setarea `inside_macro_definition = true;`

- `.ENDM`
  Nu are parametrii. Marchează finalul definiției unui macro. Nu incrementează LC dar liniile ce urmează îl incrementează. Este interpretată în pasul 1 prin setarea `inside_macro_definition = false;`

MACRO

Pentru a folosi un macro, numele lui trebuie precedat de simbolul '@'. Îl face mai vizibil pentru programator și totodată mai ușor de identificat în cod. Un macro poate fi folosit inclusiv în interiorul definiției unui alt macro dar nu în propria lui definiție.

```
1   NOP: .MACRO
2       ZERO
3   .ENDM
4
5   ADD: .MACRO a b
6       b r1
7       r1 a
8       r1
9   .ENDM
10
11  @ADD x y
12  @NOP
13  @NOP
14
15  x:  .DATA 5
16  y:  .DATA 3
17  r1: .DATA 0
```

Listing 4.9: Exemplu macro

Comentarii

Suportă doar simboluri de o linie. Caracterul ';' e desemnat pentru a marca începutul unui comentariu. Totodată, din cauză că o linie are doar 256 de caractere, tot ce se află dupa acele 256 de caractere e ignorat. Textul care urmează după o instrucțiune cu 3 parametrii sau după ultimul argument al unei directive este de asemenea ignorat. Se recomandă ca toate comentariile să înceapă cu ';' și să nu se depășească niciodată 256 de caractere pe o linie, deși nu se generează o eroare.

```
1    A ;this is a comment
2    A B                                consider this to be after 256 characters
3    A B C this is also a comment
4    m1: .DATA -1 this is also a comment
```

Listing 4.10: Exemplu de comentarii

4.2.4 Exemplu cod

```
1    reset: .MACRO p
2        p
3    .ENDM
4
5    sub: .MACRO a b
6        b a
7    .ENDM
8
9    add: .MACRO a b
10       @SUB r1 b
11       @SUB a r1
12       @reset r1
13   .ENDM
14
15   MULT: .MACRO a b
16       @SUB r1 b
17       @SUB r1 one
18       @SUB r2 a
19       @RESET a
20       loop: @SUB a r2
21       one r1 loop
22       @RESET r1
23       @RESET r2
24   .ENDM
25
26   ;// FIRST CORE PROGRAM //
27   @SUB b a
28   @RESET a
29   .END
30
31   a: .DATA 4
32   b: .DATA 2
33
```

```
;-----------------------------------------------------------------

;// SECOND CORE PROGRAM //
.ORG 128
@MULT x y
@MULT x y
.END

x:  .DATA 3
y:  .DATA 2
r1: .DATA 0
r2: .DATA 0
ONE: .DATA -1

; at the end will be added symbol ZERO which is common for the cores
```

Listing 4.11: Exemplu de cod în subleq assembly

```
a b ?
a a ?
ZERO ZERO *
.DATA 4
.DATA 2
.ORG 128
y r1 ?
one r1 ?
x r2 ?
x x ?
r2 x ?
one r1 loopWJMS
r1 r1 ?
r2 r2 ?
y r1 ?
one r1 ?
x r2 ?
x x ?
r2 x ?
one r1 loop4VZY
r1 r1 ?
r2 r2 ?
ZERO ZERO *
.DATA 3
.DATA 2
.DATA 0
.DATA 0
.DATA -1
.DATA 0
```

Listing 4.12: Conținutul fișierului intermediar pentru exemplul dat

4.2.5 Raportarea erorilor

În limbaj sunt documentate 12 tipuri de erori care pot să apară Acestea sunt:

- `ERROR_INVALID_DATA_PARAM` – parametru invalid pentru directiva `.DATA`, acceptă doar numere și nume valide de etichete
- `ERROR_INVALID_SYMBOL_NAME` – nume invalid de etichetă, trebuie să înceapă cu o literă și poate conține cifre, nu poate conține simboluri speciale, maxim 16 caractere
- `ERROR_UNKNOWN_DIRECTIVE` – directivă necunoscută, nu se află în lista de directive
- `ERROR_MULTIPLY_DEFINED_LABEL` – o etichetă nu are voie să fie definită de mai multe ori
- `ERROR_UNDEFINED_SYMBOL` – o etichetă sau un macro e folosit fără a fi definit undeva în cod
- `ERROR_INTERNAL_SYMBOL_REDEFINED` – `ZERO` e singurul simbol de care se ocupă *assembler*-ul, e o etichetă către o zonă de memorie ce mereu conține valoarea 0x00, nu poate fi redefinită
- `ERROR_SYMBOL_ZERO_READONLY` – la adresa simbolului `ZERO` este mereu valoarea 0, nu este permisă schimbarea valorii
- `ERROR_MACRO_INSIDE_MACRO` – un macro nu poate fi definit în interiorul unui macro
- `ERROR_ENDM_OUTSIDE_MACRO` – `.ENDM` trebuie să fie precedat de directiva `.MACRO`
- `ERROR_MACRO_NAME_MISSING` – o linie ce definește un macro trebuie să aibă o etichetă, aceasta devine numele macro-ului
- `ERROR_TOO_FEW_ARGUMENTS` – la folosirea unui macro au fost furnizate prea putine argumente
- `ERROR_TOO_MANY_ARGUMENTS` – la folosirea unui macro au fost furnizate prea multe argumente

5. UTILIZAREA SISTEMULUI

5.1 UTILIZAREA PROCESORULUI

Fișierele VHDL trebuie încărcate într-un mediu de simulare după care se simulează entitatea `CPU_TB`. Dacă se aplică modificările necesare proiectul poate fi încărcat pe o placă FPGA.

5.2 UTILIZAREA *ASSEMBLER*-ULUI

Nume

subleqasm – *assembler* pentru arhitectura SUBLEQ

Rezumat

subleqasm *infile* [-o outfile] [-s size] [-w wordsize]

Descriere

Traduce un fișier de cod sursă într-un fișier binar executabil pe hardware dedicat cu arhitectură SUBLEQ

Opțiuni

- -o, –output <file> – numele fișierului binar generat. Dacă nu e precizat atunci fișierul se va numi ‘a.bin’.
- -s, –size <number> – dimensiunea minimă în cuvinte a fișierului binar generat. Dacă dimensiunea finală a fișierului e mai mică decât dimensiunea minimă atunci se scrie valoarea 0x00 până se atinge valoarea minimă. Dacă nu e precizat atunci dimensiunea minimă e 0.
- -w, –word <1–8> – dimensiunea în octeți a cuvântului. Dacă nu e precizat atunci dimensiunea este 8 octeți (64 biți).

6. CONCLUZII

În acest proiect am realizat un procesor cu 2 core-uri cu arhitectura SUBLEQ și un *assembler* pentru a scrie cod cu care să fie testat procesorul. Cele 2 core-uri execută programe diferite pentru a face circuitul mai simplu. Am ales să fac ceva cu totul nou pentru mine și să lucrez cu o tehnologie cu care nu sunt familiar. Din cauza lipsei mele de experiență în VHDL unele aspecte au luat mai mult timp decât mă asteptam și a trebuit să abandonez niște detalii dar în final toate punctele cheie au fost atinse. în același timp *assembler*-ul este mai complex decât era inițial în plan să fie. Implementarea memoriei cache și a macro-urilor au fost cele mai interesante dar și dificile sarcini ale proiectului. De fapt scrierea documentației a fost cea mai grea.

Pentru îmbunătățirea proiectului s-a luat în considerare schimbarea mapării memoriei cache în mapare asociativă pe seturi și implementarea politicii de înlocuire LRU. De asemenea sistemul de operare DawnOS prezintă o listă de specificații tehnice care dacă sunt implementate procesorul va fi în stare să ruleze sistemul de operare. Circuitul nu a fost încărcat pe o placă FPGA dar ținta a fost mereu doar partea de simulare.

Assembler-ul conține multe funcționalități menite să ajute programatorul în scrierea programelor atâta timp cât totul e scris într-un singur fișier. Pentru viitor pot fi implementate funcții, ceea ce ar permite crearea unui *entry-point*, precum e funcția *main* în C/C++. De asemenea un *loader* va permite compilarea mai multor fișiere sursă înt-un singur fișier binar executabil. Astfel proiecte complexe si biblioteci refolosibile pot fi dezvoltate. Ulterior poate fi implementat și un limbaj de nivel înalt.

Acestea fiind spuse, voi continua lucrul la acest proiect deoarece a fost o experiență foarte distractivă și educativă dar și pentru că este ceva ce nu mi-aș fi putut imagina că voi realiza când am început studiile de licență. În concluzie acest proiect este unul reușit.

Bibliografie

[1] Ultrascale architecture configurable logic block. Technical report, Xiling Inc., 2017.

[2] Intel® 64 and ia-32 architecturessoftware developer's manual. Technical report, Intel®, 2022.

[3] Adrià Aguilà, Marc Marí, Antoni Navarro, and Kevin Sala. Inkel^{TM} pentwice multicore processor, 2017.

[4] Ryan Donohue. Synchronization in digital logic circuits. Suport de curs, Stanford University.

[5] Danijela Efnusheva, Ana Cholakoska, and Aristotel Tentov. A survey of different approaches for overcoming the processor-memory bottleneck. *International Journal of Computer Science and Information Technology*, 9(2), 2017. Department of Computer Science and Engineering, Faculty of Electrical Engineering and Information Technologies, Skopje, Macedonia.

[6] Geri. Dawn - operating system for subleq architecture. `http://users.atw.hu/gerigeri/DawnOS/index.html`, 2021. Online, accesat 17/11/2021.

[7] Eugen Gurban. Memoria cache. Suport de curs pentru Sisteme Multiprocesor, UPT, 2022.

[8] Oleg Mazonka and Alex Kolodin. A simple multi-processor computer based on subleq, 2011.

[9] Oleg Mazonka, Nektarios Georgios Tsoutsos, and Michail Maniatakos. Cryptoleq: A heterogeneous abstract machine for encrypted and unencrypted computation. *IEEE Transactions on Information Forensics and Security*, 11(9):2123–2138, 2016.

[10] Szymon Rusinkiewicz. Finite state machines (fsms) and rams and inner workings of cpus. Suport de curs, Princeton University, 2010.

[11] David Salomon. *Assemblers And Loaders*, chapter 1–4. Ellis Horwood Ltd, Cooper Street, Chichester, PO19 1EB, 1 edition, 1993.

[12] `esolangs.org`. Addleq. `https://esolangs.org/wiki/Addleq`, 2021. Online, accesat 10/11/2021.

[13] `esolangs.org`. Befunge. `https://esolangs.org/wiki/Befunge`, 2021. Online, accesat 10/11/2021.

[14] `esolangs.org`. Bitbitjump. `https://esolangs.org/wiki/BitBitJump`, 2021. Online, accesat 10/11/2021.

[15] esolangs.org. brainfuck. https://esolangs.org/wiki/Brainfuck, 2021. Online, accesat 10/11/2021.

[16] esolangs.org. Cryptoleq. https://esolangs.org/wiki/Cryptoleq, 2021. Online, accesat 10/11/2021.

[17] esolangs.org. Djn oisc. https://esolangs.org/wiki/DJN_OISC, 2021. Online, accesat 10/11/2021.

[18] esolangs.org. Esoteric programming language. https://esolangs.org/wiki/Esoteric_programming_language, 2021. Online, accesat 10/11/2021.

[19] esolangs.org. Flipjump. https://esolangs.org/wiki/FlipJump, 2021. Online, accesat 10/11/2021.

[20] esolangs.org. Intercal. https://esolangs.org/wiki/INTERCAL, 2021. Online, accesat 10/11/2021.

[21] esolangs.org. Oisc. https://esolangs.org/wiki/OISC, 2021. Online, accesat 10/11/2021.

[22] esolangs.org. P1eq. https://esolangs.org/wiki/P1eq, 2021. Online, accesat 10/11/2021.

[23] esolangs.org. Subleq. https://esolangs.org/wiki/Subleq, 2021. Online, accesat 10/11/2021.

[24] esolangs.org. Toga computer. https://esolangs.org/wiki/TOGA_computer, 2021. Online, accesat 10/11/2021.

[25] sites.google.com. Cache memory. https://sites.google.com/site/cachememory2011/memory-hierarchy, 2011. Online, accesat 14/05/2022.

[26] Maneesh Vidhate. An exploration of lexical analysis and lex analyzer generator. Manipal University, Dubai, 2020.

[27] Tom Murphy VII. Harder drive: Hard drives we didn't want or need. 2022.

Listă de figuri

Listă de abrevieri

ADDLEQ ADD and branch if Less than or EQal to 0. 7, 13

ALU Arithmetic and Logic Unit. 23

CISC Complex Instruction Set Computer. 5, 8, 18

DJN Decrement and Jump if Not zero. 13

FORTRAN FORmula TRANslation. 5

FPGA Field-Programmable Gate Array. 22, 37, 38

ICMP Internet Control Message Protocol. 19

INTERCAL Compiler Language With No Pronounceable Acronym. 11

LC Location Counter. 22, 31, 32, 33

LRU Least recently used. 38

OISC One Instruction Set Computer. 2, 5, 6, 7, 8, 12, 13

P1EQ Plus 1 and brach if EQal. 7, 13

RISC Reduced Instruction Set Computer. 5, 18

SPI Serial Peripheral Interface. 7

SUBLEQ SUBstract and branch if Less than or EQual to 0. 5, 6, 7, 8, 13, 22, 23, 37, 38

TCP Transmission Control Protocol. 20

TOGA TOGgle And branch if the result is true. 12

TTA Transport Triggered Architecture. 12

URISC Ultimate Reduced Instruction Set Computer. 12

VHDL Very High Speed Integrated Circuit Hardware Description Language. 7, 22, 37, 38

VI Valid-Invalid. 10, 41