SZAKDOLGOZAT

Végh Sándor Zsombor

Debrecen

2010

Debreceni Egyetem

Informatika Kar

Egymagos processzorok feladatvégrehajtási mechanizmusainak vizsgálata

Témavezető:

Dr. Végh János

Egyetemi tanár

Készítette:

Végh Sándor Zsombor

Mérnök Informatikus

Debrecen

2010

Tartalomjegyzék

1 Bevezető

Eddigi egyetemi éveim alatt sok mindennel foglalkoztam, saját szórakoztatásomra szabadidőmet különféle, aktuálisan tanultakkal kapcsolatos programok írására fordítottam. Habár ezen programok közül néhány érdemes lett volna arra, hogy szakdolgozatom témájául válasszam, végül úgy döntöttem, hogy egy Végh János tanár úr által ajánlott téma kidolgozásába fogok. Egy cseh egyetemen folytatott fejlesztésbe lehetett volna bekapcsolódni, melyek célja a LEON3 szoftprocesszor [3] mikroszálas változatának kidolgozása volt.

Először az egyszerű, módosítatlan processzor működését szerettem volna megérteni, így nekifogtam az üzembe helyezésének, ami hosszas próbálkozások után végül a megfelelő szintézer program hiányában meghiúsult. Ezt követően azonban a TSIM szimulátor felhasználásával sikerült begyakorolnom az eredeti processzor assembly nyelvének használatát, hogy később csak az újszerű programfelépítésre kelljen összpontosítani.

Amikor azonban konkrétan a mikroszálasítás vizsgálatába fogtam, rá kellett jönnöm, hogy ezen technológia megértéséhez kulcsfontosságú, hogy ismerjem a többi lehetséges megoldást, és el tudjam helyezni ezt köztük. Így tehát az időm jelentős részét a különböző processzorokban alkalmazott megoldások megértésére szántam.

Végül úgy döntöttem, hogy szakdolgozatomban a bennem ilyesformán kialakult összképet foglalom össze, ezáltal segítve azokat, akik hozzám hasonlóan egy processzortechnológia megértésére vállalkoznak.

2 Sorrendi utasítás-végrehajtás

2.1 Alapelvek kialakulása

2.1.1 Algoritmizált numerikus módszerek iránti igény megjelenése

A matematika absztrakt tudomány, mely szinte az emberiség megjelenése óta folyamatosan fejlődik. Kezdetben egyértelmű célja a valós világ tulajdonságainak számszerű leírása volt, ám hamarosan felismerték, hogy egy kutatható, felfedezésre váró absztrakt világ jött létre ezáltal. A valóság egyes elemeinek vizsgálata révén fokozatosan olyan axiómarendszert alakítottak ki, mely az így létrejött, felesleges elemektől mentes, tisztán konzisztens rendszer és a valós világ közt kapcsot képez: a valós világ elemei ezáltal matematikailag leírhatóak lettek, a matematikai modell pedig az eredeti példánytól függetlenül is vizsgálhatóvá vált.

A 20. század elején már megfelelő gyakorlati eszközrendszer állt rendelkezésre ahhoz, hogy a valós világ elemeit ilyen módon, számszerűen kutassák. Ez azonban az analitikus vizsgálatok komoly korlátai miatt egyre bonyolultabb, és egyre hatékonyabb numerikus eszközrendszert igényelt. A hatékonyság növelése kétféleképpen valósulhat meg: gyorsabban interpretálható numerikus algoritmusok kidolgozásával, illetve ezen algoritmusok gyors, automatikus végrehajtása révén.

A kezdeti, mechanikai megoldásokon alapuló automaták nagy segítségnek bizonyultak a különböző számítások elvégzésében. A 20. század elején megjelentek alternatívaként analóg elektronikai, később digitális elektronikai társaik is.

Olyan eszközöket tehát viszonylag egyszerűen létre lehet hozni, melyek egy-egy matematikai művelet megoldására alkalmasak, ám a 20. század elején megjelent az igény arra, hogy komplett matematikai algoritmusok végrehajtását is automatizálni lehessen.

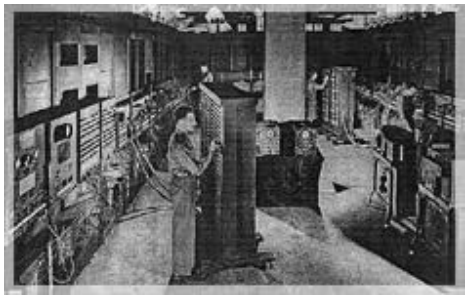
2.1.2 ENIAC - Electronic Numerical Integrator And Computer

Az első, történelmileg jelentős szerepet játszó számítógép az ENIAC volt, melyet (feltételezhetően a „világbéke” jegyében), a hidrogénbomba tervezésekor alkalmaztak először.

A legnagyobb újdonsága az volt, hogy tisztán digitális elektronikai megoldásokat alkalmazott, így nagyságrendekkel gyorsabb volt mechanikai megoldásokat is alkalmazó társainál.

További különlegessége, hogy biztosította a hiperalgoritmus végrehajtásának lehetőségét: feltételes ugró utasítást implementált, mely segítségével a korábban kiszámított eredményektől függően választhatta meg a programozó a következő végrehajtandó utasítást, ezáltal biztosítva a kívánt rugalmasságot: bármely matematikai algoritmus automatikus végrehajtására alkalmas volt [2].

Az ENIAC egymástól független modulokból épült fel, melyek általános célú buszokon keresztül kommunikáltak egymással. A végrehajtandó algoritmus megadásának szerves részét képezte ezen egységek újraszervezése. Az alkotóelemek összeköttetéseinek módosítása révén azonban tetszőleges probléma megoldására képes volt, csak a memória mérete szabott korlátot a megoldandó feladat komplexitásának, tehát Turing-teljes számítógép volt. [2]



1. kép - ENIAC

A programozástechnikai szempontból legjelentősebb moduljai a következők voltak [2]:

- Ütemező (Cycling Unit): a modulokat szinkronizálja megfelelő órajelek előállításával.
- Master Programmer: az ugró utasítás funkcionalitását megvalósító modul. Ez biztosítja a ciklusok, ugrások, szubrutinok végrehajtásának lehetőségét.

- Akkumulátor: két tíz jegyű decimális szám összeadására, kivonására, illetve egy szám tárolására alkalmas. Több ilyen típusú modult tartalmaz a rendszer. A fogaskerekes megoldás ötletét alapul véve az egyes számjegyeket tíz állapotú számlálókban tárolja, összeadásnál és kivonásnál a számláló átfordulásakor generált impulzust használja fel a következő helyiértéken álló számjegy léptetésére.
  Ilyen egységek felhasználásával bármely matematikai alapművelet megvalósítható, megfelelően szervezve őket párhuzamos működés érhető el.

Ezen kívül rendelkezett többek között egy inicializáló egységgel, mely a programvégrehajtás kezdetét és végét koordinálja, illetve lyukkártya-olvasóval és íróval.

Habár Turing-teljességéből adódóan rugalmas számítógép volt, programozása igen nagy odafigyelést igényelt. A modulok optimális szervezésének megállapítása, majd a program megírása heteket vett igénybe, ráadásul a modulok fizikai átszervezése is több napig tartott. Hibakeresési célból lehetőséget biztosított azonban a program lépésenkénti végrehajtására. [2]

2.1.3 EDVAC - Electronic Discrete Variable Automatic Computer

Az ENIAC programozása igen komplex feladat volt, így nagy sebessége ellenére sem építettek belőle több példányt. [2]

Az EDVAC tervezésekor az elsődleges cél a programozhatóság leegyszerűsítése volt. Felszerelték egy csak olvasható programmemóriával, a felprogramozáshoz elég volt ezen memória tartalmának a megváltoztatása, nem volt szükség a modulok átszervezésére. Az akkumulátorok dedikált funkciókat kaptak: megjelent a programszámláló, a matematikai műveletek operandusai ként funkcionáló akkumulátor, illetve a címregiszter. A többi akkumulátor adatmemória szerepet töltött be. [1]

Habár ezen újítások jelentős sebességcsökkenéshez vezettek, hisz megszűnt a fizikailag párhuzamos feladatvégrehajtás lehetősége, sokkal könnyebben és gyorsabban programozhatóvá vált a rendszer. A sebességcsökkenést ráadásul alig lehetett észrevenni, hisz elsősorban a lassú perifériák szabtak gátat a gyors működésnek. [2]

2.1.4 Neumann elvek

Neumann János az EDVAC-al kapcsolatos konzultációk eredménye ként írt egy piszkozatot, melyet az ENIAC építőinek munkájára alapozott, és melyben összefoglalta a számítógép építésekor szem előtt tartandó szempontokat. [4] Felismerte, hogy a még gyerek cipőben tipegő számítástechnikában az elsődleges cél az áttekinthető, egyszerűen programozható számítógép megtervezése. Jegyzetében leírtak híresültek el Neumann elvek ként, és mindaddig alapelveknek számítottak a számítógép tervezésben, amíg a technológia fejlődése lehetővé nem tette összetettebb rendszerek építését.

Neumann elvek: [4]

- Soros utasítás-végrehajtás: az utasítások végrehajtása időben egymás után kell hogy történjen. Elvetette tehát az ENIAC-ban meglévő párhuzamos végrehajtás lehetőségét, így mind a hardver felépítése, mind a rendszer programozása egyszerűbbé vált.
- Bináris számrendszer használata: a digitális elektronikai megvalósítás miatt sokkal célszerűbb a technológia által natívan támogatott bináris számrendszer használata, mint az ENIAC-ban alkalmazott decimális számrendszeré.
- Belső memória (operatív tár) használata a program és az adatok tárolására: az utasítások gyors kiolvasása, és az egyszerű programozhatóság egyértelmű előnyt jelentett az egyéb megoldásokkal szemben.

- Teljesen elektronikus működés: az ENIAC rámutatott arra, hogy mekkora sebességnövekedéssel jár az, ha kiiktatunk minden mechanikai megoldást a rendszerből.
- Széles körű felhasználhatóság: a korszak technológiai lehetőségei mellett igen nagy beruházás volt egy számítógép megépítése, így fontos volt annak újra felhasználhatósága.
- Központi vezérlőegység alkalmazása: a rendszer egyes moduljai közti kapcsolatokat nem a felprogramozáskor kell kialakítani, így elengedhetetlen egy - a teljes rendszer működését koordináló egység beiktatása.

Neumann felismerte, hogy az egyre komplexebb feladatok megoldásának gátja a kis memóriakapacitás volt. A numerikus módszerek ugyanis igen nagy mennyiségben állítottak elő közbenső eredményeket, melyek gyors hozzáférhetősége jelentősen növelte a program futásának sebességét. [1]

2.2 CISC – Complex Instruction Set Computing

2.2.1 Kétlépcsős feladatvégrehajtás

A számítógépek megjelenését kiváltó törekvés az volt, hogy minél egyszerűbben lehessen alkalmazni a matematikai modellek vizsgálatakor különböző numerikus megoldásokat. Az automatikus végrehajtás lehetőségének megteremtésével az EDVAC igen jelentős lépést tett ebbe az irányba. A módszerek alkalmazása így mindössze a megoldandó feladat megfelelő reprezentációjának megadására egyszerűsödött, onnantól kezdve automatikusan zajlott a végrehajtás. Az alkalmazás hatékonyságát tekintve tehát a szűk keresztmetszetnek a megfelelő formában történő algoritmus-leírás bizonyult.

Ez akkor vált jelentős problémává, amikor az egyre összetettebb algoritmusok leírása során felmerült a kérdés, hogy kihagyható-e a precíz, elemi eszközöket alkalmazó matematikai modell megadása a programozás folyamatából. Ekkor ugyanis kiszélesedik a szemantikai szakadék (semantic gap): igen jelentős különbség van a természetes nyelvek és a processzor utasításkészletének nyelvi eszközrendszere közt, így nehéz egy, az egyikkel leírt algoritmust a másik nyelvre fordítani.

Ezen probléma megoldására igyekeztek a processzor utasításkészletét az emberi gondolkodásmódhoz igazítani: így egyszerűbben leírhatóak lettek a természetes nyelven megfogalmazott algoritmusok gépi utasításokkal. Ez azonban a CU (Control Unit) által értelmezendő utasítások számának növelését igényelte. A kezdeti, minden végrehajtandó utasítás számára huzalozott logikát tartalmazó architektúrák tervezése így egyre bonyolultabb feladattá vált. [6] Felmerült tehát a kérdés, hogy lehet-e növelni az utasításkészlet összetettségét a hardver egyszerű felépítésének megtartásával.

A megoldást végül egy kétszintű feladatvégrehajtó rendszer jelentette. A processzorba integráltak egy mikrokód memóriát, mely egyszerűen dekódolható mikroutasításokkal leírt mikroprogramokat tartalmazott. A gépi utasítás beolvasása után

így a vezérlőegység egyszerűen elindította a megfelelő mikroprogramot, mely dekódolta az utasítás paramétereit, és végrehajtotta azt. [5]

A mikroprogramok alkalmazása igen fontos szemantikai hidat képezett a processzor által értelmezett utasítások, és a fizikai megvalósítás között. Az utasításkészlet a programozók igényeinek, a hardver pedig a technikai adottságoknak megfelelően alakíthatóvá vált.

2.2.2 Rugalmas utasításkészlet

A kétszintű architektúra rendeltetése az volt, hogy szemantikailag gazdag utasításokat biztosítson a programozók számára. Egy-egy utasítás igen összetett feladatot láthatott el: végrehajtása során több memória- és aritmetikai műveletet is végezhetett, mikroprogram-ciklusokat hajthatott végre. Egyes utasítások több száz mikroutasítás végrehajtását takarhatták.

Az assembly nyelveket használók segítésére különböző, a magasszintű vezérlési szerkezetek megvalósítását támogató utasítások jöttek létre. Később olyan utasításokat is alkalmaztak, melyek a fordítóprogramok logikáját követték. Ezek távol álltak ugyan a programozók gondolkodásmódjától, ám fordítóval fordított programok esetén hatékonyabb kódot eredményeztek. [5]

A szemantikailag gazdag utasításkészlet emellett nagy kódsűrűséget (program density) eredményezett. Ez különösen nagy előnyt jelentett a lassú hozzáférésű memóriák korszakában: a programok futtatása kevesebb memória-hozzáférést igényelt. Az egyes, nagy funkcionalitást lefedő utasítások végrehajtása ugyanis már az integrált, gyors hozzáférésű mikroprogram-memóriából zajlott. [5]

Az utasításkészlet hardvertől független, tetszőleges alakíthatósága miatt jól skálázhatóvá vált a processzor viselkedése is. Egyazon utasításkészlet implementálásával elérhető volt a különböző gyártók processzorainak bináris

kompatibilitása. Amennyiben a mikroprogramtár írható memóriában kapott helyet (WISC - Writeable Instruction Set Computer), a mikroprogramok helyszínen történő felülírásával tetszőleges kódolású bináris program futtatására alkalmas lehetett a processzor. [7]

Egyes speciálisan felhasznált belső regiszterek segítségével az operációs rendszerek által gyakran használt funkciókat lehetett implementálni a processzorra: az egyes utasítások futtatását különböző privilégiumokhoz lehetett kötni, le lehetett kérdezni a processzortól saját típusát, paramétereit, stb...

2.2.3 Szerkezeti előnyök

Az egyes mikroutasítások akár a különböző vezérlővonalak konkrét értékeit is definiálhatták meghatározott bitjeikben (horizontális mikrokód), de egyszerű tömörítő eljárással kódolt mikroutasításokat is alkalmaztak (vertikális mikrokód). Ez utóbbi esetben szükséges volt egy utasításdekódoló kombinációs logikai hálózat a mikroutasítások végrehajtásához, ám jobban kihasználta a lassú elérésű memóriák korszakában kincsnek számító gyors elérésű mikroprogram-memóriát. [7]

Lényegében tehát egy igen egyszerű felépítésű processzor emulálta egy bonyolultabb rendszer működését. Ezáltal egyazon funkcionális egységet különböző feladatok megoldására különbözőképpen használhatták fel, például ugyanazt az aritmetikai- és logikai egységet használhatták a memóriacímek kiszámítására, melyet a matematikai műveletek elvégzésére használtak. Hasonlóképpen különböző célokra használhattak fel egy adott regisztert, stb. [7]

A belső, emuláló processzort a végletekig lehetett egyszerűsíteni, így szükség esetén igen kevés funkcionális egységből is fel lehetett építeni a rendszert, sokkal egyszerűbb szerkezetű lett huzalozott társainál. [7] A kisebb és egyszerűbb felépítésű rendszer másik előnye, hogy sokkal jobban lehetett optimalizálni működési sebesség szempontjából.

Pusztán a mikrokód módosításával meg tudták változtatni a processzor viselkedését. Az egyes utasítások mikroprogramjainak módosításával azok funkcionalitásának megváltoztatása mellett a mikroprogramok optimalizálásával gyorsítható volt a processzor működése. A mikroprogramok módosítása a processzor szerkezeti egységeiben felfedezett hibák elfedésére is alkalmas volt a hibás egységek kiiktatásával, vagy a kritikus felhasználásuk szoftveres elkerülésével. [7]

2.2.4 Az architektúra korlátai

A CISC által biztosított komplex viselkedésnek igen sok előnye mellett árnyoldala is volt.

A kétszintű feladatvégrehajtás kezdetben jó megoldásnak bizonyult ugyan, de a memóriák sebességének, méretének és árának kedvező alakulása (Moore törvény) miatt később már nem okozott jelentős sebességnövekedést a gyors mikroprogram-memória.

A mikroprogramozás miatt ellenben az egyes utasítások futása változó órajelszámot igényelt. A valósidejű alkalmazások térhódítása miatt azonban olyan architektúrára volt szükség, melyen egyszerűen meghatározható a programok futási ideje. Ez a kiszámíthatatlanság nehezítette a különböző pipeline technikák alkalmazását is. [7]

A másik probléma az elvárt bináris visszafele kompatibilitás volt. Egyes utasítások a nagy bonyolultságú mikroprogramok miatt gyakran lassabban hajtódtak végre, mint az ugyanazon feladatot ellátó, egyszerűbb utasításokból felépülő utasítássorozatok. Emellett egyes, speciális funkciókat biztosító utasítások nem voltak kellőképpen kihasználva. Az újabb generációs processzorokból azonban a bináris kompatibilitás jegyében ezeket nem lehetett kihagyni. [7]

2.3 RISC – Reduced Instruction Set Computing

2.3.1 Új igények megjelenése

A CISC filozófia a korábbi tapasztalatokra építve elsődlegesnek tartotta a programvégrehajtás jól elhatárolható absztrakciós szintekre tagolását. Ennek segítségével tehette lehetővé a programozók számára a kényelmes assembly programozást, és a programok hordozhatóságát, miközben a hardver elemekkel gazdaságosan bánhatott.

Az időközben fejlődő fordítótechnológia azonban sokkal hatékonyabban hidalta át a gépi kód és a megírandó program közti szemantikai szakadékot. A fordítóprogramok segítségével ugyanis tetszőleges programozási nyelven megírt programot tetszőlegesen egyszerű utasításkészletre lehetett fordítani.

A program futtatásának több absztrakciós szintre történő bontása természetesen egyes esetekben továbbra is hasznos lehetett, például biztonságkritikus alkalmazásoknál. A digitális automaták térhódításával azonban egyre több helyen alkalmaztak processzorokat egyes folyamatok automatikus, intelligens vezérlésére.

Tekintsünk példaként egy virtuális gépen futó Java alkalmazás által vezérelt mikrohullámú sütőt. Ekkor a következő absztrakciós szintek vannak jelen az egyébként igen egyszerű program futtatása során:

- Java virtuális számítógép
- amit a virtuális hardveren futó operációs rendszer értelmez
- amit a host operációs rendszer futtat
- ami a processzornak ad gépi kódú utasításokat
- ami mikroprogramokat indít el
- ami mikroutasításokat futtat
- amik egy kombinációs hálón keresztül állítják be a processzor vezérlővonalait

Jól látható, hogy a nagy intenzitással teret hódító beágyazott rendszerek esetén, ahol a biztonságos programfuttatás és a bináris kompatibilitás nélkülözhető, ennél sokkal egyszerűbb futtatási mechanizmus is elegendő volt. Ezek a rendszerek más követelményrendszert támasztottak a feladatvégrehajtó egységekkel szemben: kis méret, egyszerű felépítés, energiatakarékosság, nagy számítási sebesség, kiszámítható futási idő.

2.3.2 A mikrokód kiiktatása

Az egyszerűsítés ötlete adott volt: a szemantikai hidat képező mikroprogramozott szerkezetet felcserélték egy egyszerű huzalozott processzorral, melyre összetett programok segítségével generálhattak futtatható kódot.

Ezzel jelentősen leegyszerűsíthető volt a fizikai rendszer felépítése, míg a fordítóprogramok összetettsége növekedett. Az ő feladatuk lett, hogy a magasszintű nyelven adott programot a processzor egyedi, elemi utasításokat tartalmazó utastáskészletére fordítsák, és optimalizálják a processzor adottságainak figyelembe vételével, például betöltsék a SPARC processzornál bevezetett késleltetési rést (delay slot). A programok egyedfejlődése így bővült egy igen jelentős fázissal, mely hatással van a felépítésre, a futási időre és a tárigényre: a kezdeti programírás- és programfuttatás közt megjelenik a fordítási fázis.

A fordítási fázis mint szemantikai híd megjelenése szükségtelenné tette, hogy a processzor utasításkészlete támogassa az assembly programok írását. A cél kevés, egyszerűen dekódolható, így gyorsan végrehajtható utasítás támogatása lett. Emiatt a vezérlőegység leegyszerűsödött, a processzorok tervezésének, fejlesztésének időtartama jelentősen lerövidült. [11]

Míg a CISC processzorok esetén a teljes rendszer akár 50%-át elfoglalhatta a vezérlőegység, addig a RISC processzorok esetén ez 6% körül mozgott. [11] A gyors tervezhetőségnek és sok felszabaduló erőforrásnak köszönhetően a processzorok

célorientáltan tervezhetők lettek: a szükséges utasítások mellé beiktathatók voltak az bizonyos feladatok megoldását gyorsító, specializált utasítások. Ez jelentős lépés volt a CISC processzorok univerzalitás felé törekvéséhez képest, melynek eredménye, hogy azok célorientált utasításai gyakran teljesen kihasználatlanok maradtak, az azokat megvalósító logika sokszor lényegében az erőforrások pazarlását jelentette. [9]

Az egyszerű felépítés további előnye, hogy az alacsonyabb áramköri sűrűség miatt magasabb lehetett a működési sebesség, és gyorsabb lapkát (GaAs) is lehetett használni. [11] Az utasítások lényegében a mikrokódolt technológiában alkalmazott vertikális mikrokódhoz hasonlítottak: fix hosszúságúak voltak, egy egyszerű kombinációs logikai hálózat segítségével lehetett őket dekódolni.

2.3.3 Felértékelődő regisztertár

Az utasítások ilyen mértékű leegyszerűsítése a címzési módok újragondolását igényelte. Lehetetlené tette ugyanis a közvetlenül memóriabeli elemeken végzett aritmetikai műveletek alkalmazását. A memóriaelérés az explicit betöltési/írási (load/store) műveletekre korlátozódott. [10] A regiszterek kis számának, így kis címszélességének köszönhetően lehetővé vált a háromcímes utasítások használata, mely jelentősen növelte az utasításkészlet rugalmasságát, hatékonyságát. [11]

Felértékelődött tehát a regisztertár szerepe. A főtárhoz fordulások számának csökkentése érdekében sok, a program futása során zajló adatforgalmat biztosító regisztert volt érdemes biztosítani. A vezérlőegység leegyszerűsítése révén felszabadult erőforrások egy részét felhasználva nagyméretű háromutas elérésű regisztertárat lehetett előállítani, ám a háromcímes, fix méretű utasítások alkalmazása korlátot szabott a címezhető regiszterek számának. [11] Ezen problémára az jelentette a megoldást, hogy mindig csak egy részét tették címezhetővé az egyébként nagyméretű regisztertömbnek.

Több megoldás is született arra vonatkozóan, hogy mi alapján tegyék címezhetővé a teljes regisztertár egy részét: [11]

- Regiszterbank (register banking) - a teljes regisztertárat azonos méretű, nem átlapolódó részekre osztották fel, ezek lettek a “bank”-ok. Egy-egy ilyen bank mérete 2 valamely hatványa lett, az aktuális bank kezdőcímét a CBP (Current Bank Pointer) jelölte ki.
- Ablaktechnika (register windowing) - a regiszterbankhoz hasonlóan részekre (window) bontották a regisztertárat, ám ezek az ablakok átlapolhatók voltak. Az aktuális ablak kezdőcímét a CWP (Current Window Pointer) határozta meg.
- Blokktechnika (register blocking) - tetszőleges méretű, átlapolható részekre (block) bontotta a regisztertömböt, az aktuálisan használt blokk kezdőcímét a CBP (Current Block Pointer) jelölte ki.
- Automatikus csere (dribble-back) - egy minimálisan két, nem átlapolódó blokkra bontott regisztertárral dolgozó rendszer. Az aktuálisan használt tárat rendeltetése szerint, regisztertömb ként kezelte a processzor, míg a háttérben a passzív tár regisztereinek értékét egy lassabb, nagyméretű tárolóval szinkronizálta.

3 Konzervatív sebességnövelési megoldások

3.1 Bevezető

A Neumann elvek egyszerűsége biztosította a számítástechnika korai szakaszában a processzorok igen látványos fejlődését. A félvezető-technológia méret- és sebességbeli határaihoz közelítve azonban túlzottan erős megkötésnek bizonyult az utasítások disztingvált, tisztán lineáris végrehajtása, hiszen így a processzor órajele szigorú elvi határt szabott az adott idő alatt végrehajtható utasítások számának.

A feladatvégrehajtás gyorsítása érdekében különböző trükkökhöz folyamodtak a processzorgyártók, miközben a processzorok kívülről nézve látszólag továbbra is a Neumann elveknek megfelelően működtek.

Ebben a fejezetben azon technikák közül tekintem át a legjelentősebbeket, melyeket a Neumann elveknek megfelelően felépített programot végrehajtó processzorok gyorsítására alkalmaztak. Lényegük az utasításszintű párhuzamosítás (Instruction Level Parallelism, ILP) lehetőségének valósidejű felismerése, kihasználása volt.

3.2 Memóriakezelés

3.2.1 Harvard architektúra

A kezdeti számítógépek a Neumann elveknek megfelelően egyetlen memóriaegységet használtak a programok és az adatok tárolására. A processzorok sebességének ugrásszerű növekedése révén azonban a relatíve nagy késleltetési idejű memóriák sebessége komoly korlátozó tényezővé vált.

Ezen problémára a megoldást a Harvard architektúra alapgondolata biztosította. Amennyiben a módosítandó adatokat és a programokat külön memóriában tároltuk, az utasítás beolvasása az adatmemóriához fordulással egy időben is történhetett. [12]

Szükségtelenné tette azt is, hogy a használt memóriák paramétereikben megegyezzenek. A szóhossz, címszélesség, implementációs technológia, időzítési paraméterek és a memória-struktúra is igazodhatott a tárolt tartalomhoz: a programmemória lehetett csak olvasható, és szóhossza tetszés szerint igazodhatott a processzor utasításkészletéhez, míg az adatmemória továbbra is írható-olvasható maradt, és processzor-szóhossz méretű blokkokban lehetett címezni [12].

Egyes, különösen sebességkritikus esetekben (valósidejű adatfeldolgozást végző DSP-k esetén) előfordult, hogy az adatmemória is több különböző memóriaegységben kapott helyet, melyeket különböző címterekben, egymással párhuzamosan lehetett elérni, tovább növelve ezzel a memóriaműveletek időbeli hatékonyságát. [12]

Habár az eredeti Harvard architektúra szerint adatokat csak az adatmemóriában tárolhatunk, a modern megvalósítások a további párhuzamosítás érdekében biztosítanak lehetőséget arra, hogy közvetlenül a programmemóriából használjunk fel konstans értékeket azok előzetes adatmemóriába másolása nélkül. [12] Ezáltal egyetlen órajel alatt hajthatók végre a megfelelő, két, memóriából olvasott paraméterrel dolgozó utasítások.

3.2.2 Gyorsítótár

A tisztán Harvard architektúra igen egyszerű szerkezeti felépítést követelt meg, elveszítették azonban a Neumann elvek által nyújtott rugalmas, univerzalitást támogató programszemléletet. [12]

A két megoldás előnyeit azonban jól lehetett ötvözni: kisméretű, ám igen gyors hozzáférésű gyorsítótárba (cache) duplikálhatták a főtár releváns lokációinak tartalmát, melyekhez gyorsan, egymástól függetlenül férhettek hozzá. Ezzel a megoldással a leggyakoribb, hogy a program kódjáról készítettek másolatot, ám írható- olvasható cache alkalmazásával az adatmemóriáról is készíthettek gyors hozzáférésű replikát. [12] Az így kapott rendszer a Neumann architektúra rugalmasságát mutatta, hiszen egyetlen főtár tartalmával dolgozott, ám a gyors másolatok konkurens hozzáférésének köszönhetően a Harvard architektúra sebességét kínálta.

A gyorsítótár az egyes memóriabeli elemekről egymástól függetlenül tartalmazott hozzáférhető másolatot. Ennek köszönhetően hozzáféréskor kombinációs hálóval kellett ellenőrizni, hogy az adott memóriabejegyzés szerepelt-e a cache-ben. Általában a következő három megoldás közül alkalmazták valamelyiket ezen ellenőrzéshez [14]:

- Teljesen asszociatív gyorsítótár (Fully Associative Cache) - egymástól független blokkbejegyzéseket tartalmazott, minden alkalommal az összes bejegyzéshez hasonlította a memóriacímet. Rugalmas megoldás, jó találati arányt lehetett vele elérni, ám igen erőforrás-igényes volt.
- Közvetlen leképezésű gyorsítótár (Direct Mapping Cache) - a cím alsó néhány bitje alapján kapott helyet a bejegyzés. Ekkor csak az így meghatározott bejegyzéshez letárolt címet kellett megvizsgálni.
- Csoport-asszociatív gyorsítótár (Set Associative Cache) - a kettő kombinációja. A bejegyzések csoportokra voltak bontva, a cím felső néhány bitje alapján eldőlt, hogy mely csoportra képeződhetett le az adott bejegyzés, ám azon belül

bárhova. A kisszámú összehasonlítás miatt viszonylag gyors, jó találati arányú megoldás volt.

Memória-hozzáféréskor először ellenőrizte a gyorsítótár, hogy az adott címhez tartalmazott-e már bejegyzést. Amennyiben igen (cache hit), egyből hozzáférhettek a cache tartalmához. Ha azonban az adott lokáció nem volt a gyorsítótárban (cache miss), új bejegyzést allokált, belemásolta a memóriacímen elérhető tartalmat, ezt követően folytathatták a munkát. Ez azonban lassú folyamat volt. Az új bejegyzés allokálásakor így perdöntő volt a jó heurisztika alkalmazása az eldobandó bejegyzés kijelölésére. Leggyakrabban a legutoljára használt bejegyzés eltávolítását választották. [14]

Habár a gyorsítótárazási megoldások alkalmazása a Neumann architektúra rugalmasságát biztosította, voltak kedvezőtlen vonatkozásai. Mind kis költség- és erőforrás-igényű, egyszerű rendszerek esetén, mind a kiszámítható futásidőt igénylő valósidejű jelfeldolgozó processzorok esetén célszerűbb volt az egyszerű, predesztinált végrehajtási idejű Harvard architektúra alkalmazása. [12]

3.3 Utasítások egyidejű végrehajtása

3.3.1 Csővezeték

A processzorok igen sokat változtak, fejlődtek a kezdeti megoldásokhoz képest, ami azonban az egyes egységeik összetettségének növekedését eredményezte. Márpedig minél összetettebb volt egy kombinációs logikai hálózat, annál nagyobb lett a felhasznált elektronikai elemek késleltetéséből adódó válaszideje. Egyes esetekben, például összeadó, szorzó áramkörök esetén ezt a válaszidőt megfelelően felhasznált extra logikával csökkenteni lehetett, ám csak kis mértékben, és nagy költségek árán.

A gyorsítás érdekében vessünk egy pillantást a kombinációs logikai hálókra, melyek a bemeneteikre kapcsolt bitek értéke alapján, aszinkron módon állítják elő a

kimeneti értékeket! A nagy válaszidőt az eredményezi, hogy az egymás után kapcsolt kapuk késleltetési ideje összeadódik. Átalakíthatjuk a rendszert szinkron működésű soros logikai hálózattá egyszerűen azáltal, hogy szakaszokra (lépcsőkre) bontjuk, melyek tárolókon keresztül kapcsolódhatnak egymáshoz.

Az így kapott hálózat annyi órajel-periódus alatt állítja elő a kívánt kimenetet, ahány lépcsőre bontottuk a végrehajtást. Azt, hogy milyen frekvencián üzemeltethető a rendszer, a legnagyobb válaszidejű lépcsője határozza meg. A soros működtetés eléréséhez beiktatott tárolók késleltetése is lassítja az egységet, így jól láthatóan mindenképpen nő az egy eredmény kiszámításához szükséges idő.

Hatékonysága abban rejlik, hogy nagy órajel-frekvenciával üzemeltethető, és több eredmény kiszámítását végzi párhuzamosan, ugyanis órajelenként új bemenet feldolgozásába foghat, mely alapján előállított eredmény a lépcsőkön végiglépkedve adott időn belül megjelenik a kimeneten.

3.3.2 Utasítás-csővezeték

A Neumann elvű processzor feladatvégrehajtási mechanizmusa igen elvont volt, jól elkülöníthető fázisokra lehetett bontani. Ilyen fázisok lehettek például az utasítás betöltése, az operandusok meghatározása, a kijelölt művelet végrehajtása. Ezen fázisok mindegyikét összetett logikai hálózat valósította meg, ám az utasításciklus során ezek közül egyszerre mindig csak egy volt aktív. [16]

Az alkalmazott megoldás, hogy a kombinációs logikai hálók csővezetékelésének mintájára minden órajelnél új utasítás végrehajtásába fogott a processzor, ezáltal átlapolva az utasítások végrehajtását: minden időpillanatban több, különböző fázisban lévő utasítás végrehajtása zajlott párhuzamosan. Ezáltal adott működési frekvencia mellett növelni lehetett a rendszer feladatvégrehajtási sebességét. A legtöbb mai processzorgyártó legalább kétlépcsős csővezetéket alkalmaz. Az Intel Pentium 4 húsz

lépcsőre bontotta a feladatvégrehajtást, de akad több mint száz lépcsős megoldás is (például a Xelerator X10q). [15]

A kombinációs logikai hálókból kialakított rendszerekhez képest azonban az utasítás-végrehajtás csővezetékelése összetettebb feladatnak bizonyult. A processzorok lelke ugyanis abban rejlett, hogy az egyes végrehajtott utasítások hatással lehettek a következő utasítás megválasztására, vagy annak végrehajtására. Márpedig egy utasítás nem léphetett olyan végrehajtási fázisba, amely függött egy másik, még végrehajtás alatt álló utasítás hatásától. Ilyen esetekben késleltetni kellett az adott utasítás előrehaladását azáltal, hogy várakozási ciklust, buborékot (bubble) indítottak el a csővezetékben az utasítás továbbléptetése helyett. [15]

A buborékok beiktatása lassította a program végrehajtását, így kiszámíthatatlan lett a futási idő. Akár az is előfordulhatott, hogy egyes programrészek futtatása során a csővezetéknek mindig csak egy lépcsője volt aktív, a többi csak buborékot tartalmazott, ilyenkor lassabb is lehetett a rendszer működése, mint egy ugyanolyan, ám nem csővezetékelt processzoré.

A hosszú csővezetékek alkalmazásának további mellékhatása volt, hogy egyes, erre nem felkészített önmódosító programok hibás futását eredményezte. Ilyenkor ugyanis előfordult, hogy a módosításkor a megfelelő utasítás végrehajtása már megkezdődött. [15]

3.3.3 Superscalar

A processzorral szemben elvárás volt, hogy egy szekvenciális utasítássorozat elemeit dolgozza fel sorrendhelyesen. A csővezeték megoldásokból látható, hogy az egymás utáni utasítások feldolgozása átlapolható, ám a megfelelő eredmény elérése érdekében az egyes utasítások által kijelölt műveleteket elkülönülten kell végrehajtani.

A superscalar processzorok azt használták ki, hogy az egymás utáni utasítások gyakran egymástól teljesen függetlenek voltak, akár párhuzamosan is végre lehetett őket hajtani, és ez nem változtatott az általuk kiváltott hatáson. Ezen utasításszintű párhuzamosíthatóság kihasználására a processzort több redundáns funkcionális egységgel (például több ALU, FPU) látták el, melyek fizikailag párhuzamosan hajthattak végre több utasítást. [17]

Mivel továbbra is ragaszkodtak a Neumann elvek által diktált egyszerű programfelépítéshez, az utasításszintű párhuzamosítás lehetőségét a processzornak kellett felismerni, ezt a feladatot az utasítás-elosztó (instruction dispatcher) látta el. Beolvasta a memóriából az utasításokat, eldöntötte, melyeket lehetett közülük párhuzamosan végrehajtani, és azokat különböző funkcionális egységekhez irányította. [17]

Ahhoz, hogy hatékonyan tudja kiválasztani a processzor a soron következő utasítások közül, hogy melyek voltak egyszerre végrehajthatók, természetesen egyszerre több utasítást kellett beolvasnia és megvizsgálnia, nagy utasításablakra (Instruction Window, IW) volt szükség.

A párhuzamosan működő funkcionális egységek miatt a superscalar processzoroknak több, független pipeline-ja is lehetett, az utasítás-elosztó ezekbe adagolhatta a végrehajtandó utasításokat egyazon utasítássorból. A processzor így egy órajel alatt akár több utasítás végrehajtását is megkezdhette, így nagyobb végrehajtási átlagsebességet elérve. [17]

A kezdeti architektúrák két-három redundáns egységet alkalmaztak, ám a mai változatok akár négy ALU-val, két FPU-val és egy SIMD egységgel is rendelkezhetnek. Ennél több redundancia alkalmazása azonban már nem célszerű, hatékony kihasználásukhoz ugyanis igen összetett, pontos utasítás-elosztóra van szükség. Kihasználhatóságuk mértéke függ a végrehajtott program utasításszintű

párhuzamosíthatóságától is. Problémát jelenthetnek például az előző utasítás által előállított eredményt felhasználó, illetve ugró utasítások. [17]

3.4 Spekuláció

3.4.1 A jóslás szükségessége

A csővezeték alkalmazásával nagymértékben gyorsították a program futását, hiszen a függőséget nem tartalmazó utasítások feldolgozását átlapolták, így növelve az utasítás-végrehajtás átlagsebességét. A gyorsítás mértéke azonban nagyban függött a futtatott utasítássorozat szerkezetétől, benne rejlő függőségektől.

Különösen komoly problémát okoztak a feltételes, illetve a közvetett ugrások, ahol az utasítás-csővezetéknek várakozni kellett mindaddig, amíg az ugró utasítás le nem futott, és be nem állította a megfelelő programmemória-címet.

Egyes, két lépcsős csővezetéket tartalmazó RISC processzorok (például SPARC) ezen probléma áthidalására nemes egyszerűséggel késleltetési résnek (delay slot) nevezték az ugró utasítást követő programmemória blokkot, és az ott található utasítást mindig beolvasták, és végre is hajtották, ezáltal megőrizve a csővezeték folyamatosságát. A késleltetés explicit kérésére itt NOP utasítást adhattunk ki, ám adott volt a programozó számára a lehetőség, hogy sebességkritikus programok esetén itt is hasznos utasítást hajtson végre.

Hosszabb csővezeték alkalmazásakor azonban erre nem volt lehetőség. Ilyenkor célszerű volt megjósolni a következő utasítás címét, és ott folytatni a végrehajtást. Amennyiben a jóslat helyesnek bizonyult, az utasítás-csővezeték folyamatossága megmaradt, így gyorsítva a program futását. Jól kihasználható mechanizmusokat alapoztak pusztán az ugró utasítás kódjának vizsgálatára: [18]

- Miután sok ciklust tartalmaztak a programok, melyeknek gyors futása komoly hatással lehet a futási időre, ezek támogatása érdekében azt jósolták, hogy csak a visszafele, tehát kisebb programmemória címre ugrások feltétele teljesül.
- Mivel a teljes programkörnyezet ismerete sokat javíthat a jóslás minőségén, lehetőséget adtak a programozóknak is arra, hogy a különböző utasításkódok formájában a programkódba ültessenek egyszerű, a célszerű ugrásirányt definiáló információkat.

3.4.2 Alkalmazott mechanizmusok

Amennyiben az ugró utasítás befejezése után hibásnak bizonyult a számított jóslat, el kellett vetni az addigra részben végrehajtott utasításokat, és a megfelelő helyen kellett folytatni a program végrehajtását. Az így elvesztett idő mértéke függött a lépcsők számától, melyeken átesett az ugró utasítás beolvasásától a végrehajtásáig. Ez modern mikroprocesszorok esetén könnyen elérhette a 10-20 órajel-periódust. Minél hosszabb volt tehát az utasítás-csővezeték, annál nagyobb szerepet kapott a jó hatásfokkal működő jósló mechanizmus. [18]

Miután a program felépítése szempontjából továbbra is a Neumann architektúra egyszerűségét szerették volna nyújtani a gyártók, az összetett, kódba integrált jóslások szóba sem jöhettek. Mivel azonban az ugró utasítások többsége valamilyen kiszámítható rendszer szerint dolgozott, például időben periodikus volt a működése, vagy az esetek többségében azonos módon viselkedett, ugrási előzményük nyomon követésével jó hatásfokú jóslásokat lehetett biztosítani.

Az előzmények alapján történő jóslást megvalósító áramkör a jósló áramkör (branch predictor) volt. A néhány legutóbb futtatott releváns utasítás ugrási előzményeiről informatív statisztikát tárolt, mely alapján később jó hatásfokú jóslást adott a következő utasítás címére vonatkozóan. [18]

Mivel a processzor univerzális feladatmegoldó képessége a hiperalgoritmusok végrehajtásán alapult, igen nagy számban kellett ugrásokat végrehajtani, így fontos volt a lehető legjobb tárolt információ – felhasznált erőforrás arányú módszer alkalmazása az ugráselőzmények statisztikájának tárolásánál.

- A legegyszerűbb esetben minden feltételes ugró utasításhoz annak címe alapján, egyetlen biten tárolta, hogy az a legutolsó futtatása során ugrott, vagy sem. Ekkor a következő alkalommal azt jósolhatta, hogy ugyanúgy fog viselkedni. Egyes ciklusok esetén jó hatásfokkal működhetett. [18]
- Az egybites megoldás rossz tulajdonsága volt, hogy az egy-egy alkalmat leszámítva egységesen működő utasításoknál, például ciklusleszárásnál túlkompenzálta a változást: kétszer is rosszul jósolt egymás után. Ennek orvoslására bevezették a kétbites szaturáló számlálót, ami a nagyon valószínűtlen, valószínűtlen, valószínű, és nagyon valószínű állapotok közt lépkedhetett az előző ugrás eredménye alapján. Ekkor az egyszeri eltérő viselkedés nem változtat a jóslaton. [18]
- Amennyiben egy utasítás minden második, vagy minden harmadik alkalommal ugrott, a szaturáló számláló igen rossz jóslási arányt tanúsított. Emiatt változtattak a módszeren: nem magukhoz az utasításokhoz, hanem azok ugrási előzménymintáihoz rendeltek szaturáló számlálókat (kétszintű adaptív jósló). Hárombites előzmény alkalmazása esetén például egy ugró utasításhoz nyolc számláló tartozott, és mindig az ugró utasítás aktuális hárombites előzményének megfelelő bejegyzéssel dolgozott a jósló. Ezen megoldás alkalmazásával hamar fel lehetett ismerni bizonyos periodikus mintákat. [18]
- Az idáig tárgyalt megoldások minden utasításhoz külön előzményeket tároltak, így azonban rejtve marad a különböző utasítások viselkedése közti esetleges korreláció. Ezen tulajdonságok kihasználására egyes esetekben egyetlen kétszintű adaptív jóslót alkalmaztak, amelyet minden ugró utasítás, annak programmemória címétől függetlenül használhatott (gshare). Ez a módszer nagy táblaméretet igényelt, ám az 1989-es SPEC sebességmérés eredményei alapján megközelíthette akár a lokális jóslók hatékonyságát is. [18]

3.4.3 Speciális viselkedésminták kezelése

Voltak olyan egyszerű viselkedésminták, melyek felismeréséhez nem volt szükség a fenti, nagy erőforrás-igényű megoldások alkalmazására. Ilyen lehetett például az előírt lépésszámú ciklus ugró utasítása, mely csakis minden N-edik alkalommal ugrott / nem ugrott.

Az indirekt ugró utasítások célcímének meghatározása legtöbbször túl összetett feladat ahhoz, mintsem hogy huzalozott logikával megoldható lett volna. A leggyakoribb megoldás volt azt jósolni, hogy ugyanoda ugrik, ahova azt legutóbb tette.

Igen speciális ugrásfajtának bizonyult a függvényvisszatérés, melynek célcíme mindig elérhető volt a visszatérési verem tetején. Mivel a verem olvasása túl időigényes folyamat lett volna, ennek kihasználásához egy gyors hozzáférésű, kisméretű verem-puffert alkalmaztak, melynek tartalma a függvényhívások- és visszatérések alkalmával automatikusan frissült. [18]

A jó hatékonysággal működő jósló áramkörök igen összetettek lehettek, ennek megfelelően nem voltak elég gyorsak az eredmény egyetlen órajel alatti előállításához. Ilyenkor az utasítás-csővezeték telítettségének megőrzése érdekében egy egyszerűbb megoldás alkalmazásával előzetes jóslást végeztek, mely alapján megkezdhették a spekulatív végrehajtást. Amikor néhány órajellel később előállt a precízebb jóslat, ezt felülbírálhatták. [18]

3.5 Out of Order Execution

3.5.1 Függőségtípusok

A processzorok egyetlen feladata az volt, hogy a kapott utasítás-szekvencia elemeit sorrendhelyesen végrehajtsák. Mint azt a korábbiakban láthattuk, az utasítás-csővezeték nagyban felgyorsíthatta ezen folyamatot, ám egyes esetekben lehetetlen volt egy utasítás feldolgozásának megkezdése az őt megelőzőek lezárása előtt.

A spekuláció bevezetésével az ugró utasítások utáni késleltetés szükségtelenné vált, de az utasítások végrehajtásának folyamatosságát gyakran meg kellett szakítani egyes, várakozást megkívánó események miatt. Ilyen lehetett például egy memória- vagy periféria-hozzáférés.

Egy-egy ilyen esemény időtartama alatt igen sok utasítást lehetett volna végrehajtani, ám ezt az utasítás-szekvencia elemeinek esetleges függőségei miatt csak nagy körültekintéssel lehetett megtenni. Függőség akkor állhatott fenn, ha két egymás utáni utasítás egyazon regisztert, illetve perifériát hivatkozta. Ezt a következőképpen tehették:

- Olvasás az olvasás után (Read after Read, RaR) – az utasítások csak olvasták az adott regisztert, nem módosítottak annak tartalmán. Ekkor tetszőleges sorrendben végre lehetett őket hajtani.
- Írás az írás után (Write after Write, WaW) – egy adott regiszternek mindig az utolsó oda írt értéket kellett visszaadnia. Ezen esetben mindkét utasítás felülírta az adott regiszter tartalmát. Ha végrehajtásukat felcserélték, nem a megfelelő eredmény marad meg a célregiszterben.
- Írás az olvasás után (Write after Read, WaR) – miután kiolvasták a regiszter tartalmát, a következő utasításban újra szerették volna hasznosítani, új adattal feltölteni. Ha felcserélték a két utasítás végrehajtását, akkor először felülírták a regiszterben tárolt fontos információt, még mielőtt az olvasó utasítás hozzáférhetett volna.

- Olvasás az írás után (Read after Write, RaW) – a regiszter korábbi tartalma érdektelen szemét, a cél az volt, hogy egy új értékkel írják felül, melyet a következő utasítás hasznosíthatott. Ekkor felcserélésük után az olvasó utasítás a korábbi szemetet olvasta ki, és csak ezt követően került az értelmes adat a regiszterbe. Mint azt a továbbiakban látni fogjuk, ez az egyetlen függőség, mely jellegéből adódóan nem feloldható.

A nem sorrendi feladatvégrehajtás (Out of Order Execution, OoO) alapgondolata alapján mialatt egy utasítás saját operandusainak megjelenésére várakozott, a fenti függőségek figyelembe vételével a processzor kereshetett, és feldolgozhatott olyan utasításokat, melyek végrehajthatóak voltak a program működésének befolyásolása nélkül. [19]

3.5.2 Olvasás utáni olvasás – Scoreboard

Az első, legegyszerűbb megoldás a scoreboard alkalmazása volt. A processzor minden utasítás beolvasásakor ellenőrizte az általa használt regiszterek függőségi viszonyait, és csak akkor engedte tovább, ha már nem volt konfliktus az előzőleg elindított, még nem végrehajtott utasításokkal. [20]

- Beolvasás – ekkor ellenőrizte a rendszer az utasítás által hivatkozott regisztereket.
- Várakozott az utasítás egészen addig, amíg WaW függőség állt fenn, illetve foglaltak voltak a megfelelő hardver elemek.
- Várakozott az által használt összes olvasott regiszter tartalmának rendelkezésre állásáig, így figyelembe véve a RaW függőségeket.
- Végrehajtásra került az utasítás, ezt követően a scoreboard értesítést kapott az előrehaladásról.
- Az eredmény célregiszterbe másolása késleltetve volt egészen addig, amíg minden, az adott regisztert olvasó utasítás be nem fejezte olvasási fázisát. Így akadályozták meg a WaR hazard bekövetkeztét.

Mai szemmel nézve ez a megoldás egyfajta rugalmas sorrendi végrehajtás volt, hiszen megállt az első írás utáni írás típusú kapcsolatnál. Ennek köszönhető, hogy habár a függőségek hiánya esetén időnként nem sorrendben hajtott végre egyes utasításokat, szigorúan sorrendben zárta le azokat. [19]

3.5.3 Írás az írás után – Visszatérési sor

Annak érdekében, hogy egy utasítás várakozása alatt minél nagyobb részt tudjanak végrehajtani a szekvenciából, kellően rugalmas, jó hatásfokú nem sorrendi végrehajtó módszerre volt szükség. Amellett tehát, hogy a RaR utasításminták egymástól független végrehajthatóságát kiaknázták, további kapcsolattípusok nem sorrendi végrehajtását is lehetővé kellett tenniük.

A WaW függőség feloldására lehetővé tették, hogy az eredmények előállítási sorrendje különbözzön az utasítások szekvenciabeli sorrendjétől. Ehhez bevezettek egy köztes tárolót, melybe az utasítások végrehajtási fázisukban elhelyezhették az eredményeiket. Innen megfelelő sorrendben kerülhettek az adatok a célregiszterekbe. [19]

A várakozási sor bevezetésével a következő lépésekre bontották az utasítások feldolgozását: [19]

- Beolvasás
- A beolvasott utasítás egy utasítássorba (instruction buffer / queue) került.
- Itt várakozott, amíg input operandusai elérhetőek nem lettek, ezáltal elkerülve a RaW hazardot.
- Ezt követően a megfelelő funkcionális egységhez került, mely megkezdte végrehajtását.
- A végrehajtás befejezésekor a kiírandó érték a visszatérési sorba (retire stage) került.

- A WaW hazard elkerülése érdekében itt várakozott addig, míg az összes korábbi utasítás eredménye át nem került a regisztertömbbe, és csak ezt követően lehetett a célregiszterbe másolni.

Amennyiben jóslással együtt alkalmazták, előfordulhatott, hogy az ugró utasítás végrehajtási fázisba érkezésekor már befejeződött egyes, spekulatív módon megkezdett utasítások végrehajtása. Ezen eredmények elvetése is a várakozási sor feladata volt. [19]

3.5.4 Írás az olvasás után – Regiszter átnevezés

Az írás az olvasás után típusú kapcsolat valójában csak álfüggőség, legtöbbször mindössze azért állt fenn, mert a programok előszeretettel hasznosították újra igen gyakran ugyanazon regisztereket. Ez a függőségtípus így fordítási időben is feloldható lett volna pusztán azáltal, hogy az eredmények tárolására kihasználták az összes, processzor által kínált regisztert.

Ahogy nőtt a sebességbeli különbség a processzor és a memóriák közt, úgy egyre jelentősebb szerepet kapott a nem sorrendi végrehajtás, hiszen ezáltal rengeteg műveletet hajthattak végre a memóriára várakozás időtartama alatt. Fontos lett tehát minden feloldható függőség kihasználása.

Miután azonban a cél továbbra is az egyszerű Neumann architektúra szimulálása volt, nem bízhatták a programozóra, illetve fordítóra a problémát, futási idejű megoldást kellett alkalmazniuk. Az írás késleltetése helyett bevezethettek erre a célra egy új regisztert, ezáltal mind az új, mind a régi érték hivatkozható maradt. Ehhez összetett regisztertárra volt szükség, mely több regisztert tartalmazott, mint amennyit az architektúra definiált. Az egyes regisztereknek ráadásul címkézhetőnek kellett lenni. [19]

Amint minden, a régi értéket olvasó utasítás olvasási fázisa befejeződött, a megfelelő regisztert el lehetett vetni, újra lehetett hasznosítani.

4 Szakítás a Neumann elvekkel

4.1 Bevezető

A processzorgyártók által alkalmazott sebességnövelési megoldások, például a nem sorrendi feladatvégrehajtás, nagyban gyorsították a programok futását, ám cserébe komoly erőforrás-áldozatokat igényeltek. Ennek oka, hogy áramköri elemek segítségével, futási időben határozták meg az utasításszintű párhuzamosítás lehetőségét, illetve a lehetséges ugrási célcímeket.

Az így meghatározható sebességnövelési lehetőségek többsége már a gépi kód előállításakor ismert, mindezidáig azonban túlzottan ragaszkodtunk a Neumann architektúra által előírt egyszerű programszerkezethez.

Miután a szilikon technológia határaihoz közelítve a végsőkig kiaknáztuk a Neumann elvek által javallott végrehajtási mechanizmus lehetőségeit, a további fejlődéshez új paradigmák kidolgozására, vizsgálatára van szükség.

Rengeteg erőforrás takarítható meg, egyszerűbb felépítés és jobban optimalizált futás érhető el pusztán azáltal, hogy az eddig futási időben végzett utasításelemzési feladatokat a processzor huzalozott logikája helyett a fordítóprogrammal, fordítási időben végeztetjük el.

Ebben a fejezetben azon technikák közül tekintek át néhányat, melyek az iménti problémák megoldását a fordítóprogramokra bízzák.

4.2 Mikroszálas feladatvégrehajtás

4.2.1 Időosztásos futtatás

A RISC processzorok csővezetékelésének előnye abban rejlik, hogy az utasításokat átlapolva hajtják végre. Sok esetben azonban meg kell szakítanunk a csővezeték folyamatosságát. Egy ugró utasítás után például nem olvashatjuk be a sorozat következő elemét, amíg nem ismerjük az általa beállított programmemória-címet. Hasonlóképpen várakoztatnunk kell a rendszert, amennyiben nagy késleltetéssel hajtódik végre egy utasítás, például egy memóriaelérés.

Mint a korábbiakban láthattuk, ezen problémákra valósidejű megoldások születtek, ahol a processzor futási időben elemzi az utasításokat, és azokat nem sorrendi módon hajtja végre, illetve valószínűségi alapon folytatja a végrehajtást egy ugró utasítás beolvasása után. Ezen megoldások jó hatásfokkal dolgoznak ugyan, de igen sok áramköri erőforrást használnak fel, és nehezen tervezhetővé teszik a processzorokat.

A jelenleg tárgyalt megoldás alapgondolata, hogy tekintsük a programot egyetlen lineáris utasításfolyam helyett párhuzamosan futtatható mikroszálak összességének. A kód, melyet a processzorral futtatni kezdünk, lesz a fő szál (main thread), kezdetben az egyetlen mikroszál. A program futása során definiálhatunk további mikroszálakat, melyek időosztásosan futnak. [21]

A módszer sebességnövelő hatása abban rejlik, hogy ha várakozást kiváltó eseménnyel, például memóriaeléréssel találkozunk, várakozás helyett automatikusan átváltunk egy olyan szál futtatására, mely arra készen áll, anélkül, hogy buborékot iktatnánk a csővezetékbe. Míg egy szál adatra várakozik, egy másik fut. A mikroszálak így átveszik a nem sorrendi feladatvégrehajtás szerepét. [21] Szükségtelenné teszik az általa igényelt nagy utasításablak jelenlétét is, hiszen elég mindig a következő utasítás beolvasása.

Akkor is automatikus kontextusváltás történik, amikor feltételes, vagy közvetett ugró utasítást olvasunk be. Így tehát egy másik szálon dolgozhatunk, mialatt a csővezetéken végiglépdelve kiértékelődik az adott szál következő utasításának címe. Ez a jóslást alkalmazó megoldáshoz képest egyrészt azért előnyös, mert kevés extra erőforrást igényel, másrészt sokkal kevésbé lesz alkalmazásfüggő a programok futási ideje, hiszen nincsenek a hibás jóslásból adódó nagy késleltetések, harmadrészt pedig nem pazaroljuk az erőforrásokat arra, hogy nem megfelelő utasításokat dolgozunk fel, majd vetjük el a feldolgozás eredményét. [21]

A fordítóprogram feladata, hogy optimálisan bontsa szálakra a végrehajtandó feladatot, ezáltal biztosítva a lehetőséget a csővezeték folytonosságának megőrzésére. A mikroszál kifejezéshez híven a módszer az egyszerű kontextus-fogalma révén akár igen rövid, néhány utasítást futtató szálak hatékony kihasználását is lehetővé teszi. [21]

4.2.2 Regisztertár, szinkronizáció

A mikroszálak lelke a gyors és egyszerű kontextusváltásban rejlik. Az alkalmazott, legegyszerűbb megoldás szerint minden mikroszál ugyanazt a regisztertömböt látja, írhatja, olvashatja. [21] A fordító feladata úgy kiosztani a szálak közt a használható regisztereket, hogy azok véletlenül se használják fel egymás erőforrásait. Kellően nagyméretű regisztertömb esetén így is igen sok szál definiálható, ráadásul a köztük történő váltás mindössze a programszámláló lecserélésére egyszerűsödik.

A különböző szinkronizációs feladatokat is a közösen használt regisztertömb elemei segítségével valósítják meg. Ennek támogatására a regiszterek szinkronizációs bitekkel vannak ellátva. Így mindegyikhez állapot tartozhat, mely alapján azok írásakor, olvasásakor automatikusan mehetnek végbe a különböző szinkronizációs folyamatok: [22]

- Üres regiszter (empty) – kezdetben minden regiszter ebben a státuszban van. Amennyiben adatot írunk bele, teli státuszba kerül. Ha azonban olvasni

próbálunk belőle, az adott szál futása automatikusan fel lesz függesztve egészen addig, amíg adat nem kerül a regiszterbe.

- Várakozó regiszter (waiting) – azt jelzi, hogy egy szál várakozik arra, hogy az adott regiszterbe érték kerüljön. Ekkor tartalmazza annak a szálnak az azonosítóját, amely várakozik rá. Így amint adatot írunk bele, automatikusan tovább indíthatjuk a várakozó szálat.
- Teli regiszter (full) – adatot tartalmaz, és a megszokott módon használható. Írhatjuk és olvashatjuk anélkül, hogy megváltozna a státusza.

Az így definiált regiszterek segítségével adatfolyam-szinkronizáció valósítható meg. Amennyiben egy szál olyan adatot hivatkozik, mely még nem áll rendelkezésre, futása automatikusan fel lesz függesztve egészen addig, míg elő nem áll a kívánt adat. [22]

A csővezeték telítettségének megtartása érdekében az ütemezőnek tudnia kell, mely szálak állnak futásra készen, és melyek nem. Emiatt az egyes mikroszálakhoz is állapotjelző tartozik, mely a következő értékeket veheti fel: [22]

- Várakozó (waiting) – létrehozásakor ebbe az állapotba kerül. Azt jelzi, hogy még nincs a gyorsítótárban a szál következő utasítása. Amint a gyorsítótárba kerül a megfelelő utasítás, kész állapotba kerül.
- Kész (ready) – futásra kész állapot. Nem várakozik adatra, és a következő utasítás be van töltve a gyorsítótárba.
- Felfüggesztett (suspended) – amikor feltételes ugró utasítást hajtunk végre, vagy olyan regisztert olvasunk, melyben nincs adat, ebbe az állapotba kerül a szál. Egészen addig marad itt, amíg ki nem értékelődik az ugrási célcím, illetve adat nem kerül az olvasott regiszterbe. Ezt követően várakozó vagy kész fázisba kerül attól függően, hogy a következő utasítása a gyorítótárban van-e.
- Halott (killed) – azt jelzi, hogy a szál futása befejeződött, ám az erőforrásai még nem szabadultak fel.

- Futtatott (running) – az a szál, melynek kódját az adott pillanatban futtatja a processzor. Egyszerre mindig csak egy szál lehet ebben az állapotban.

4.2.3 Megvalósítás

Az aktív szálak a folytatási sorban (Continuation Queue, CQ) kapnak helyet, mely a következő adatokat tárolja bejegyzéseiben: [22]

- A mikroszálhoz tartozó programszámláló (Program Counter, PC) aktuális értéke. Miután minden szál ugyanazt a regiszterkészletet látja, egy adott szál kontextusa az aktuális programmemória-cím segítségével reprezentálható.
- A mikroszál első utasításának címe, a mikrokontextus kezdőcíme. Létrehozáskor megegyezik az előbbi mező értékével.
- Azon szál azonosítója, illetve azon regiszter, melyre a szál jelenleg várakozik (amennyiben várakozik). Az adatfolyam-szinkronizáció megvalósításának fontos eleme.
- Azonosító (slot number) – a szál azonosítója, mely révén hivatkozható.
- Következő szál – a várakozási sort láncolt lista segítségével valósítják meg, ez a bejegyzés tartalmazza a lista következő elemének azonosítóját.

Ahhoz, hogy a programot mikroszálak sokaságának formájában definiáljuk, az egyszálú, soros végrehajtású programok gondolatvilágától eltérő utasításkészletre van szükségünk. A szálak szinkronizálását, mint láthattuk, részben megoldhatjuk a regiszterek segítségével, ám létrehozásukra, és kezelésükre utasításokat kell biztosítani: [22]

- Cre (create) – szál létrehozása. Paraméterül kapja a létrehozni kívánt szál első utasításának programmemória béli címét. Lefoglal egy helyet a folytatási sorban,

belemásolja a programmemória címet, és beállítja a szál állapotát készre, vagy várakozóra attól függően, hogy a gyorsítótárban van-e az adott kezdőutasítás.

- Swch (switch) – ugrás a következő szálra. Ezen utasítás eredménye ként a folytatási sor következő elemét kezdi futtatni a processzor. Így kérhetjük az aktuális szál időleges felfüggesztését.
- Bsync (barrier synchronisation) – a szál futtatásának felfüggesztése egészen addig, míg az összes többi szál futása be nem fejeződött (barrier syncronization).
- Brk (break) – a fő szál kivételével az összes mikroszál lezárása.
- Kill – a szál lezárása. Killed-re állítja az állapotát, de addig nem szabadítja fel a dinamikusan foglalt erőforrásokat, amíg van őket olvasó szál. A legegyszerűbb megoldás, hogy egészen addig nem szabadítja fel az erőforrásokat, amíg van aktív szál.

4.2.4 Szál családok

A magasszintű vezérlési szerkezetek egyik sajátos esete az előírt lépésszámú ciklus. Egyes esetekben már fordítási időben is tudjuk, hogy a futóváltozó milyen értékeket fog felvenni, ráadásul akár teljesen függetlenek is lehetnek egymástól az egymást követő ismétlődések. Az előírt lépésszámú ciklus így egy módja annak, hogy hogyan adhatjuk meg egyszerre mikroszálak egy csoportját. Ezt a csoportot nevezik a szálak egy családjának.

Ugyanaz a programkód tartozik minden iterációhoz, mégis párhuzamosan szeretnénk futtatni őket. Annak érdekében, hogy saját regiszterekkel rendelkezzenek, dinamikusan kell számukra az erőforrásokat kiosztani. Ennek bevett módja, hogy a szál nem közvetlenül, hanem egy ablakon keresztül hivatkozza a regisztereket. Relatív címeket alkalmaz, az ablak kezdőcíme az adott szál egyik jellemző értéke lesz. [22]

A processzor dinamikusan hozhatja létre a szálakat, és amint egyikük befejeződött, automatikusan létrehozhatja helyette a következőt, így számukat

folyamatosan a maximumon tartva. Ez által csökkenti az esélyét annak, hogy egyszerre mindegyik várakozzon valamire.

Az iterációk közti esetleges függőségek a normál mikroszálakhoz hasonló módon, a regiszterek szinkronizációs státuszai segítségével kerülnek kezelésre.

Ahhoz, hogy dinamikusan hozzunk létre ilyen szál családokat, további utasításokra van szükségünk. Az uT-LEON3 processzorban használt szálcsalád kezelő utasításokból a fontosabbakat kiemelve láthatjuk, hogy ez jelentősen növeli a rendszer összetettségét: [23]

- Setstart – beállítja egy szálcsalád számára a futóváltozó kezdőértékét.
- Setstep – a futóváltozó léptetésekor alkalmazott lépésköz meghatározása
- Setlimit – a futóváltozó végső értékének meghatározása
- Setblock – a szálcsalád definíciója alapján a processzor automatikusan hoz létre mikroszálakat. Ezen utasítással adható meg az a határszám, hogy egyszerre hány szál legyen jelen az adott családból.
- Setplace – a szálcsalád erőforrásainak lefoglalása
- Setregs – a szálcsalád által használt regiszterablakok kezdőcímének beállítása
- Squeeze – a szálcsalád minden elemének felfüggesztése. Nem szabadítja fel az erőforrásokat, a későbbiekben visszatérhetünk a család futtatásához.

4.3 Explicit párhuzamosság

4.3.1 VLIW – Very Long Instruction Word

A egyszerű Neumann elvű számítógépek minden funkcionális egységből, így aritmetikai- és logikai, vagy lebegőpontos egységből is egyet alkalmaztak, hisz nem volt szükség több művelet egyetlen órajel alatt történő végrehajtására.

A superscalar processzorok már több funkcionális egységet láttak el egyidejűleg feladatokkal. Miután ezek futási időben mérték fel, hogy a következő utasítás végrehajtását meg lehetett-e kezdeni, illetve mely funkcionális egységhez lehetett rendelni, komoly korlátot jelentett a programok szekvenciális szemlélete.

A fordítóprogramok összetettségének növekedése, és a memória-áramkörök gyors fejlődése lehetővé tette, hogy az addig huzalozott logika segítségével valós időben megvalósított ütemezést, illetve az adathazardok kezelését átvállalják a fordítóprogramok, így születtek meg a nagyon széles utasításszavú (Very Long Instruction Word, VLIW) processzorok. [27]

A VLIW processzorok utasításai a RISC-hez hasonlóan fix méretűek. Egy-egy ilyen komplex utasítás meghatározott számú egyszerű utasítást tartalmaz, melyek egy időben hajthatók végre, különböző funkcionális egységeken. A fordítóprogram így lehetőséget kap a processzor erőforrásainak optimális kihasználására. Miután az ütemezés statikus, a továbbiakban nincs szükség a superscalar feladatmegosztásért felelős utasítás-elosztó áramkörre. [25][27]

A nem sorrendi feladatvégrehajtást, illetve superscalar feladatelosztást megvalósító áramkörök elhagyásával igen sok áramköri erőforrás szabadul fel, melyek új funkcionális egységek integrálását teszik lehetővé, ezáltal növelve a fordítóprogramok mozgásterét a program párhuzamosításában.

A processzor egyszerű felépítése és nagy sebessége miatt gyakran alkalmaznak VLIW processzorokat beágyazott rendszerekben. [25]

4.3.2 EPIC – Explicitly Parallel Instruction Computing

A VLIW processzorok alapelvei közzé tartozik, hogy statikus ütemezést alkalmaznak, és egy-egy utasításukban minden funkcionális egység feladatát definiálják. Miután azonban egyes hardver elemek, például a DRAM memóriák késleltetési ideje nem determinisztikus, a fordítási idejű ütemezés nem minden esetben megoldható. [27]

Az alapgondolat tehát önmagában sok problémát vet fel, melyek orvoslására a RISC processzorok sebességnövelésénél tárgyalt technikák egyszerűsített változatai jelenthetik a megoldást. Ezen megoldások keresztezéséből született az EPIC (Explicitly Parallel Instruction Computing).

- A komplex utasításokat utasításkötegeknek (bundle) hívjuk, melyekhez stop bitet csatolunk, ami meghatározza az előző utasításoktól való esetleges függését, ezáltal támogatja a kiszámíthatatlan késleltetési idejű utasítások végrehajtását. [27]
- Gyorsítótárat alkalmaz, melynek működését a programkód befolyásolhatja. Utasításokat tartalmaz a különböző memóriablokkok előzetes gyorsítótárazására. [27]
- Igen nagy regisztertömböt biztosít a program köztes eredményeinek tárolására, így a fordítónak ritkán kell az egyes regisztereket újrahasznosítani, így kerüli el az írás az olvasás után típusú adatfüggőségeket, szükségtelenné téve a futási idejű regiszterátnevezési megoldásokat. [27]
- Az ugró utasítások funkcionalitását több egyszerű részre bontja, azokat különálló utasításokkal megvalósítva. Külön értékeli ki az ugrás feltételét, számítja ki a

célcímet, és hajtja végre az ugrást, így egyetlen utasításköteg több alternatív ugrás közül választhat. [27]

4.4 NISC – No Instruction Set Computing

4.4.1 Utasítások nélkül

A RISC koncepció egyszerűsége abban rejlett, hogy a program-memóriából a processzor egyszerűen kódolt utasításokat olvasott ki, melyek egy egyszerű dekódoló kombinációs logikai hálózaton keresztülhaladva közvetlenül határozták meg az adatút vezérlését szolgáló vezérlővonalak állapotát (vertikális nanokód). A programok utasításszekvencia szerű felépítése azonban egy idő után komoly korlátozó tényezővé vált.

A VLIW processzorok ezt módosítva biztosítanak lehetőséget arra, hogy a fordító statikus ütemezéssel, optimálisan használhassa fel az erőforrásokat. Ennek elérése érdekében komplex utasításokat alkalmaznak, egyszerre több funkcionális egység számára egyszerű, vertikálisan kódolt utasítások formájában definiálják a végrehajtandó utasítást.

Mindkét megoldás esetén fontos szempont az assemblyben programozhatóság, holott az egyre összetettebb fordítóprogramok akár automatikus szintézist is lehetővé tesznek. Az utasításkészlet nélküli számítás (No Instruction Set Computing, NISC) erről az oldalról közelíti meg az ütemezés kérdését: ahelyett, hogy egyszerűen dekódolható utasításokat alkalmazna, a dekódolás fázist kihagyva minden utasítás bitjeiben közvetlenül a vezérlővonalak kívánt állapotait tárolja (horizontális nanokód). [24]

Az egyes kontroll szavak kétszer- háromszor szélesebbek, mint a RISC processzorok esetén, hisz szélességüket a processzor vezérlővonalainak száma határozza meg. A programok mérete azonban nem növekszik, ugyanis egy-egy NISC utasítás átlagosan két-három RISC utasítás funkcionalitását fedi le. [28]

4.4.2 A hardver és szoftver határainak elmosódása

A fordítónak közvetlen hozzáférése van a multiplexerekhez: az adatúthoz, a regiszterekhez. Teljesen megszűnik a programok bináris hordozhatósága, cserébe viszont a fordító tetszőleges mélységig csővezetékelheti, párhuzamosíthatja az adatutat, így korlátozások nélkül használhatja a processzor erőforrásait. A leegyszerűsödő vezérlőegységnek köszönhetően ráadásul nagyobb működési frekvencia érhető el. [8]

Habár a végsőkig adott az erőforrás-optimalizálás lehetősége, tetszőlegesen egyszerű fordító is alkalmazható, mely akár mindössze egy RISC assembly utasításaiban gondolkodva, azok dekódolt, NISC megfelelőiből állítja össze a programokat. Hasonlóképpen tetszőleges RISC processzort emulálhatunk egy megfelelő NISC processzor segítségével, ha a megfelelő utasításdekódoló logikán keresztül csatoljuk a programmemóriát. Az utasítások ilyen jellegű dekódolásához akár utasítástáblázat is használható. [8]

A horizontális nanokód alkalmazása miatt nincs szükség az utasításkészlet kidolgozására, az utasításdekódoló- és ütemező áramkörök megvalósítására. Igen egyszerű tehát a processzor által tartalmazott funkcionális egységek módosítása, számuk megváltoztatása. A technológia így nagyban támogatja az egyedi, adott feladathoz igazított processzorok tervezését, illetve előre megtervezett és tesztelt áramkörrészletek, a speciális funkcionalitást biztosító IP Core-ok (Intellectual Property Core, szellemi tulajdont képező áramkörtervek) alkalmazását. [24]

Miután a lefordított program a vezérlővonalak szintjén definiálja az adatutak működését, a fordító számára adott a lehetőség, hogy az áramkör viselkedését részletesen definiálja. A bináris kód pedig így közvetlenül a hardveren fut, előzetes átalakítás nélkül. A program és a processzor tehát egymás nélkül nem értelmezhető. [8]

A hardver és a szoftver közti határvonal kezd szertefoszlani: a hardver pusztán erőforrásokat biztosít, míg a szoftver feladata azok megfelelő felhasználása. Már nincs meg a Neumann architektúra által javallott elvonatkoztatás, mely szerint a processzor feladata, hogy ügyeljen a kapott utasítások helyes végrehajtására. [8]

Utasításkészlet híján a fordítóprogramot eleve fel kell készíteni az erőforrások közvetlen kihasználására, így a korábbi, RISC szemlélettől eltérően elegendő egyetlen fordítóprogram, melynél a fordítandó program megadásának szerves részét képezi a rendelkezésre álló erőforrások definiálása. [29]

4.5 Szoftprocesszorok

4.5.1 Egyszerűsödő hardvertervezés

A szoftprocesszorok megemlítése előtt vetnünk kell egy pillantást arra, hogyan vált lehetővé a megjelenésük.

Az integrált áramkörök tervezésére különböző lehetőségek állnak a gyártók rendelkezésére aszerint, hogy mennyire fontos a készülő áramkör sebessége, kiszámíthatósága, a tervezés rövid időtartama, illetve a gyártás egyszerűsége: [30]

- Megtervezhetik a készítendő integrált áramkör minden rétegének teljes fotolitográfiai maszkját (Full Custom Design), így nagy áramköri sűrűséget érhetnek el, ám igen költséges a tervezés fázis, és sok tesztelésre van szükség az áramkör elkészítése után.
- Az egységesítés nevében könyvtárakat hozhatnak létre a felhasználható áramköri elemekből (Standard Cell Design). Ezen elemeket pontos karakterisztikájukkal együtt adhatják meg, így felhasználásukkal gyorsan, biztonságosan tervezhetőek áramkörök, szimulálható azok működése. Az

elemek helyes működését a teszteken túl a sokszoros felhasználás révén a gyakorlati tapasztalat is igazolja.

- Még gyorsabb tervezés érhető el azáltal, ha a felhasználható áramköri elemeket tartalmazó rétegeket előre elkészítik (Gate Array Design). Ebben az esetben az integrált áramkör tervezése azon rétegek megtervezését jelenti, melyek kialakítják a kapcsolatokat a már beépített áramköri elemek közt. Habár egyszerű és gyors megoldás, előre definiált rétegei révén nem támogatja az egyedi elemek, IP Core-ok felhasználását. További kellemetlen vonatkozása, hogy miután tervezéskor csak az összeköttetések kialakítására van lehetőségünk, az adott áramkörhöz nem lesz optimális a felhasználható elemek elhelyezkedése.

A fejlődő szoftveripar egyre több támogatást nyújt az integrált áramkörök tervezéséhez. A kézzel történő huzalozás helyett már lehetőségünk van arra, hogy különböző, hardverek viselkedését nyelveken adjuk meg az elvárt funkcionalitást, és ez alapján megfelelő programok állítják össze, szintetizálják a kívánt áramkört. Hasonlóképpen lehetőségünk van egy kész áramkörterv viselkedésének szimulálására, így kiszűrve annak hibáit még az előállítása előtt. [30]

Az integrált áramkörök tervezési, és gyártási fázisa a következőképpen alakul: [30]

- Felvázoljuk a megoldandó problémát
- Valamilyen hardverleíró nyelven (VHDL, Verilog) leírjuk a rendszer elvárt működését. A fordítóprogram ez alapján létrehoz egy sematikus áramköri tervet. (Register Transfer Level, RTL).
- Szimulációk révén megvizsgáljuk, hogy az áramkör alkalmas-e a feladata teljesítésére.
- Logikai szintézis segítségével az RTL sémát a gyártó által kínált áramköri elemekre képezzük le. A szükséges áramköri elemeket, és a köztük kialakítandó kapcsolatokat netlist-nek nevezzük.
- A place fázisban kapnak helyet a felhasznált áramköri elemek az integrált áramkörön belül
- Végezetül a route fázisban hozzuk létre a kialakítandó kapcsolatokat.

4.5.2 Újrakonfigurálható eszközök

A Gate Array Design lényege, hogy az áramkör tervezését a már előre elkészített, és elhelyezett áramköri elemek közti kapcsolatok kialakítására redukálja. A újrakonfigurálható eszközök esetén az IC-t előre elkészítik olymódon, hogy a kapcsolatok kialakítására szolgáló rétegeket programozható kapcsolatokkal látják el. [30]

Ehhez tartomány alapú (domain based) kapcsolótáblákat alkalmaznak: ahol két vezeték metszi egymást, ott az így kapott négy vezetékrészlet közül bármely kettő közt megadhatjuk, hogy legyen-e kapcsolat. A kapcsolók állását az eszköz saját belső memóriájában tárolja, ennek írásával változtatható meg az összeköttetések rendszere, így az áramkör viselkedése. [31]

A felhasználás céljától függően alapvetően két újrakonfigurálható eszköztípust különböztetünk meg:

- CPLD (Complex Programmable Logic Block) – egyszerű felpítésű, elsősorban a különböző lábai közti kis késleltetésű, programozott kapcsolatok megvalósítását szolgálja. Gyors, diszjunktív normál formában adott logikai függvényeket megvalósító áramköri elemekből építkezhetünk. [32]
- FPGA (Field Programmable Gate Array) – összetett áramkörök megvalósítására szánt eszköz. Előre definiált rétegeiben tartalmaz gyakran használt funkcionalitásokat megvalósító áramköröket is, így szorzókat, memóriákat, órajel manipuláló áramköröket, akár analóg elemeket is. [31]

Az FPGA célja az univerzális felhasználhatóság, miszerint bármely integrált áramkör funkcionalitását meg tudja valósítani. Rugalmas építőelemei a CLB-k (Complex Logic Block), melyek valamilyen kombinációs logikai függvényt, kapukat, illetve tárolókat tartalmazhatnak. A CLB-k és a dedikált elemek használhatóak fel a kész áramkör létrehozására. [31]

Habár igen sokoldalúak, az FPGA-k kezdetben elég rossz hatásfokkal dolgoztak. Hozzávetőleg tizennyolcszor annyi területet, és hétszer annyi energiát használtak fel egy funkció megvalósításához, mint annak Full Custom Design segítségével előállított megfelelője. A sebességkülönbség körülbelül háromszoros volt. Az eltelt idő folyamán azonban mind az FPGA-k ára, mind pedig hatékonysága jelentősen javult.

A újrakonfigurálható eszközök nagy segítséget nyújtanak az áramkörtervezésben, ahol egyfajta élethű próbapanel ként lehetőséget biztosítanak az áramkörök tesztelésére, illetve a gyors hibajavításra. Nagy szerepük van azonban a beágyazott rendszerekben, és az egyedi fejlesztésű rendszerekben is, hiszen bármely célorientált hardver gyors és egyszerű létrehozását lehetővé teszik. [31]

4.5.3 Az újrakonfigurálhatóság jelentősége

Az FPGA rugalmasságával és univerzális eszközrendszerével könnyen járható utat nyitott az egyedi tervezésű hardverek előállítása felé. Míg egy összetett integrált áramkör tervezése, és fotolitográfiai maszkjainak előállítása igen sok időt vesz igénybe, és akár dollármilliókba is kerülhet [33], addig egy újrakonfigurálható eszköz lényegében plusz költség nélkül képes bármilyen áramkör kialakítására.

Az áramkörök viselkedésének leírására alkalmas nyelvek (VHDL, Verilog, LabVIEW) további, hatalmas segítséget nyújtanak a hardverek létrehozásában. Az integrált áramkör elvárt viselkedését definiálva a megfelelő programok segítségével szintetizálhatjuk az áramkör tervét, illetve az eszköz felprogramozására használható bitvektort.

A szintézis folyamata természetesen igen összetett, sok időt vesz igénybe. Míg egy alkalmazásfüggő integrált áramkör igen nagy fejlesztési idejéhez képest eltörpült az a néhány óra, melyet az áramkör szintetizálására kellett szánni, a néhány másodperc leforgása alatt felprogramozható FPGA-k használatakor ez időnként aránytalanul lassú folyamat.

Az újrakonfigurálható eszközök megjelenésének köszönhető tehát, hogy tetszőleges, nagysebességű logikai áramkör igen kis költséggel hozható létre. Általuk adott a lehetőség arra, hogy pusztán kísérleti jelleggel próbáljanak ki különböző megoldásokat, mint például a jelenleg tárgyalt, újszerű koncepciókat alkalmazó processzorokat.

A hardverleíró nyelveknek köszönhetően egyes esetekben lehetőség nyílik arra is, hogy processzoron futtatott program helyett egy az egyben célhardvert definiáljunk az adott feladat elvégzésére, illetve gyors perifériákat készítsünk az eszközben megvalósított processzorhoz.

4.5.4 Egyedi processzorok

Amennyiben a kívánt szoftver futtatására használt processzort ASIC (Application Specific Integrated Circuit) helyett egy újrakonfigurálható áramkörben implementáljuk, igen egyszerűen tudunk változtatni annak felépítésén. Költséghatékonyan definiálhatunk új, illetve cserélhetünk le korábbi funkcionális egységeket, melyek az adott programok futtatását gyorsítják.

Miután az adatutak megváltoztatása a vezérlőegység módosítását is igényli, természetesen gyakran a leghatékonyabb megoldás a már kész processzor adottságainak felhasználásával, annak nyelvi eszközrendszerére fordítani a kívánt programot, ám minél sebességkritikusabb alkalmazásról van szó, annál több hardveres támogatást lehet nyújtani az egyes algoritmusok futtatásához.

Igen rugalmas konstrukció ilyen szempontból a NISC. Nagy előnye abban rejlik, hogy miután eleve nincs utasításkészlete, így vezérlőegysége, a fordító a rendelkezésre álló adatutak adottságait felhasználva fordítja le, illetve optimalizálja a programot, így tetszőleges funkcionális egységekkel bővíthetjük a rendszert. [29]

Megfelelően összetett szintézer képes lehet arra is, hogy fordításkor ne csak a program futtatott kódját állítsa elő, hanem annak gyors futását segítő funkcionális egységeket hozzon létre, és a hozzá legjobban illő adatutakat is definiálja, ezáltal lényegében a futtatott programhoz előállítva a számára legmegfelelőbb processzort. [29]

A szoftprocesszorok használata révén tehát véglegesen megszűnik a határ a futtatott szoftver, és az azt futtató hardver közt, hiszen azok gyakran egyszerre jönnek létre, és csak együtt értelmezhetőek.

5. Összegzés

A szakdolgozatom témájának kiválasztásakor eredeti célom a mikroszálas technológia kutatása, és a LEON3 szoftprocesszor [3] mikroszálas változatának kidolgozásába való bekapcsolódás volt.

Miután sikerült közel kerülnöm ahhoz, hogy üzembe helyezzem a módosításra váró szoftprocesszort, a TSIM nevű szimulátor segítségével közelről megismerkedtem annak assembly nyelvével.

Szerkezeti felépítésének vizsgálata előtt azonban úgy döntöttem, hogy mielőtt folytatom a munkát, utánanézek a processzortervezésben alkalmazott egyéb megoldásoknak. Sok időt áldoztam a különféle alternatívák megismerésére, mígnem el tudtam helyezni köztük a vizsgált módszert. Mivel a bennem ilyesformán kialakult összképet igen hasznosnak találtam, úgy döntöttem, szakdolgozatomban ezt adom tovább.

Ennek megfelelően ebben a munkában az egymagos, regiszter alapú processzorok építése során az első tisztán elektronikus számítógép (ENIAC) óta alkalmazott megoldások közül veszem sorba a legjelentősebbeket. Mindegyik esetén szót ejtek annak alapötletéről, valamint a kialakulása miértjéről.

Az áttekintés az egymagos, regiszter alapú processzorokra terjed ki, így nem foglalkozom a programszintű párhuzamosítás, a többmagos, többprocesszoros rendszerek kérdésével. Nem mutatok be konkrét eszközöket sem, hiszen célom a módszerek miértjének, szellemiségének ismertetése, azok átfogó áttekintése.

6. Irodalomjegyzék

[1] http://en.wikipedia.org/wiki/John_von_Neumann
[2] http://en.wikipedia.org/wiki/ENIAC
[3] http://www.gaisler.com/cms/index.php?option=com_content&task=view&id=13&Itemid=53
[4] http://en.wikipedia.org/wiki/Von_Neumann_architecture
[5] http://en.wikipedia.org/wiki/Complex_instruction_set_computing
[6] http://www.laynetworks.com/CISC.htm
[7] http://en.wikipedia.org/wiki/Microcode
[8] http://www.ijest.info/docs/IJEST10-02-06-39.pdf
[9] http://www.heyrick.co.uk/assembler/riscvcisc.html
[10] http://www.inf.u-szeged.hu/projectdirs/bohusoktat/regi/szganyagok/esszek/A_RISC_jellemzoi.pdf
[11] http://www.uni-miskolc.hu/~qgerecon/hf3.html
[12] http://www.pic24micro.com/harvard_vs_von_neumann.html
[13] http://www.wordiq.com/definition/CPU_cache
[14] http://en.wikipedia.org/wiki/CPU_cache
[15] http://en.wikipedia.org/wiki/Instruction_pipeline
[16] http://en.wikipedia.org/wiki/Classic_RISC_pipeline
[17] http://en.wikipedia.org/wiki/Superscalar
[18] http://en.wikipedia.org/wiki/Branch_predictor
[19] http://en.wikipedia.org/wiki/Out-of-order_execution
[20] http://en.wikipedia.org/wiki/Scoreboarding
[21] **Micro-Threading: A New Approach To Future RISC** (2000)
Chris Jesshope, Bing Luo
[22] **Instruction-level Parallelism through microthreading - A Scalable Approach to Chip Multiprocessors** (2006)
Kostas Bousias, Nabil Hasasneh, Chris Jesshope
[23] **Instruction Set Extensions for Multi-Threading in LEON3** (2010)
M.Danek, L.Kafka, J.Sykora
[24] http://en.wikipedia.org/wiki/NISC
[25] http://en.wikipedia.org/wiki/Very_long_instruction_word
[26] http://en.wikipedia.org/wiki/Superscalar
[27] http://en.wikipedia.org/wiki/Explicitly_Parallel_Instruction_Computing
[28] http://www.ibridgenetwork.org/uci/nisc-processor-design
[29] http://www.ics.uci.edu/~nisc/
[30] http://en.wikipedia.org/wiki/Application-specific_integrated_circuit
[31] http://en.wikipedia.org/wiki/Field-programmable_gate_array
[32] http://en.wikipedia.org/wiki/CPLD
[33] http://www.mil-embedded.com/articles/id/?3664