2. Mikrokontrollera komponentes

2.1 Procesora kodols

CPU ir galvenā sastāvdaļa jebkuram mikrokontrolierim. Tas bieži vien tiek ņemts no jau eksistējošiem procesoriem. Piemēram MC68306 mikrokontrolieris no Motorola ir ar 68000 CPU. Jums jau vajadzētu būt iepazinušamies ar materiāliem šajā sadaļā no citām sadaļām, tātad šeit tikai virspusēji tiks aprakstītas svarīgākās lietas, bet netiks ieslīgts detaļās.

ARHITEKTŪRA

Pamata CPU arhitektūra ir attēlota 2.1. attēlā. Tā sastāv no datu plūsmas kas izpilda instrukcijas un no kontroles bloka kas datu plūsmai saka ko darīt.

Aritmētiski loģiskais bloks (arithmetic logic unit ALU).

CPU pamatā ir ALU, kas tiek izmantots lai veiktu skaitļošanas un darbības (AND, ADD, INC). Dažas kontroles līnijas nosaka kuras operācijas ALU vajadzētu veikt pie datu ievades. ALU saņem divus signālus un atgriež rezultātu. Mērķis un avots tiek ņemti no reģistriem kas uzglabājas atmiņā. Papildus tam ALU saglabā informāciju par rezultātiem statusa reģistrā (status register saukts arī par condition code register):

Z (nulle – zerro) : operācijas rezultāts ir 0.

N (negatīvs – negative) : operācijas rezultāts ir negatīvs, tas ir vis vērtīgākais bits (most significant bit - MSB) rezultātā tiek nomainīts uz 1.

O (pārplūde – owerflow) operācijas rezultātā radās pārplūde, tas nozīmē, ka nomainījās zīme *divu komplementācijas* operācijā.

C (pārnese – carry) : Operācijas rezultātā radās pārnese.

Divu komplementācija

Tā kā datori izmanto tikai 0 un 1 lai apzīmētu skaitļus tad rodas jautājums kā apzīmēt negatīvus skaitļus. Pamata ideja ir apgriezt visus pozitīvā skaitļa bitus pretēji lai iegūtu negatīvo skaitli. Bet šī metode ir ar nelielu negatīvo iezīmi, tādu ka nulle tiek atkārtota divreiz (tad kad visi biti ir 0 un visi biti ir 1). Tādēļ daudz labāks veids kā tos attēlot ir apgriezt pozitīvo skaitli un pievienot tam 1.

4 bitu reprezentācijā tas izskatās šādi:

1 = 0001 -> -1 = 1110+1 = 1111.

Priekš nulles:

0 = 0000 -> -0 = 1111+1 = 0000,

Tātad šādā veidā parādās tikai viena nulle. Šī reprezentācijas metode tiek saukta par divu komplementāciju.

Reģistra faili satur strādājošus CPU reģistrus. Tas var sastāvēt vai nu no vispārēja uzdevuma reģistriem (parasti 16 – 32 bet var būt arī vairāk), katrs no tiem var būt operācijas avots vai mērķis vai sastāvēt no veltītajiem reģistriem veltītie reģistri ir piemēram akumulators kas tiek izmantots priekš aritmētiskām/loģiskām operācijām vai rādītāja reģistriem kas tiek izmantoti dažās adresēšanas metodēs.

Jebkurā gadījumā CPU var izmantot operandus priekš ALU no failiem un tas var saglabāt operācijas rezultātus reģistru failos. Alternatīva tam ir operandus/rezultātus var būt no atmiņas. Tomēr atmiņas piekļūšanas laiks ir lielāks kā piekļuves laiks pie reģistriem, tātad prātīgāk būtu izmantot reģistra failus ja tas ir iespējams.

Steka rādītājs (stek pointer SP)

Steks ir daļa no secīgās atmiņas datu telpā kuru izmanto CPU lai saglabātu atpakaļadreses un iespējams arī reģistru datus. Tas ir saistīts ar komandu PUSH (novietot kaut ko stekā) un POP (noņemt kaut ko no steka). Lai saglabātu šā brīža aizpildes līmeni stekā, CPU ir speciāls reģistrs kas tiek saukts par steka rādītāju, kas norāda uz steka virsotni. Steki parasti pildās uz leju, tas ir, no augstākas atmiņas adreses uz zemāku. Tātad steka rādītājs sākas datu atmiņas beigās un tiek samazināts ar katru PUSH darbību vai palielināts ar POP darbību. Iemesls šādam steka rādītāja novietojumam ir tāds ka novietojot steka rādītāju datu atmiņas beigās jūsu mainīgie tiek novietoti datu atmiņas sākumā, tātad novietojot steka rādītāju beigās, atmiņai ir nepieciešams ilgāks laiks lai tās abas saskartos.

Diemžēl ir divi veidi kā paskaidrot uz kurieni atmiņā norāda steka rādītajs: to var uztvert kā pirmo brīvo adresi tātad PUSH vajadzētu saglabāt datus tur un tad samazināt steka rādītāju kā parādīts zīmējumā 2.2. (Atmel AVR kontrolieris izmanto steka rādītāju šādā veidā), vai arī tas var tikt saprasts kā pēdējā izmantotā adrese, tātas PUSH darbība no sākuma samazina steka rādītāju un tikai tad ievieto datus jaunajā adresē (šādu interpretāciju ir adoptējusi, piemēram, Motorola HCS12). Kopš steka rādītājs ir jāinicializē programmētājam tad ir nepieciešams noskaidrot kā kontrolieris ar to rīkojās.

Kā jau tika minēts tad kontrolieris izmanto steku gadījumos kad normālā programmas gaita ir pārtraukta un vēlāk tai jāatsākas. Tā kā atgriešanās adresei ir priekšnosacījums ka programmai kuras darbība tiek atjaunota pēc punkta pārrāvuma visi kontrolieri uz steku pārvieto vismaz atgriešanas adresi. Daži kontrolieri par saglabā reģistru datus stekā lai nodrošinātu to nepārrakstīšanos. Tas galvenokārt tiek darīts ar kontrolieru palīdzību kam ir tikai neliels skaits ar veltītajiem reģistriem.

Vadības bloks

Neņemot vērā dažas speciālas situācijas tādas kā HALT instrukcijas vai reset, tad CPU visu laiku izpilda programmas instrukcijas. Tas ir kontroles vienības uzdevums noskaidrot kuras operācijas vajadzētu izpildīt kā nākamās un sakārtot datu ceļu pareizi. Lai to izdarītu vēl viens speciāls reģistrs, programmu skaitītājs (Program counter - PC), tiek izmantots lai saglabātu nākamās programmu instrukcijas adresi. Kontroles vienība ielādē instrukciju instrukciju reģistrā, dekodē instrukciju, un uzstāda datu ceļu lai to izpildītu. Datu ceļa konfigurācija sevī iekļauj nepieciešamos datus priekš ALU (no atmiņas reģistriem), izvēlas pareizo ALU operāciju un parūpējas par to lai rezultāts tiktu ierakstīts pareizajā vietā (reģistrā vai atmiņā). Programmu

skaitītājs ir vainu palielināts lai norādītu uz nākamo instrukciju kas seko vai arī tiek ielādēta jauna adrese negaidītam gadījumam. Pēc pārlādēšanās programmu skaitītājs parasti ir $0000.

Tradicionāli kontroles vienība ir cieši saistīta, tai ir tabula kurā tiek glabātas vērtības ar to kuras kontroles līnijas ir nepieciešams izmantot lai veiktu instrukciju, kā arī diezgan sarežģīta dešifrēšanas loģika. Tas nozīmē ka ir sarežģīti nomainīt instrukcijas kas ir dotas CPU. lai atvieglotu kontroles vienības uzbūvi, Maurīcijs Vilkes atspoguļoja ka kontroles vienība ir patiesībā mazs CPU un ir spējīgs iegūt labumu pats no savām mikroinstrukcijām. Pēc viņa kontroles vienības uzbūves programmas instrukcijas tika sadalītas mikroinstrukcijās, katra no kurām veica kādu mazu daļu no visas lielās instrukcijas (tāpat kā padot pareizos reģistrus ALU). Tas protams pārvērta kontroles uzbūvi par programmēšanas uzdevumu: pievienot jaunas instrukcijas pie instrukciju kopas noveda pie instrukciju programmēšanas mikrokodā. Kā sekas tam, pēkšņi kļuva samērā viegli pievienot jaunas un sarežģītas instrukcijas, un tā rezultātā instrukciju kopas kļuva arvien lielākas un spēcīgākas. Tas noveda pie arhitektūras vārdā Complex Instruction Ser Computer (CISC). Protams spēcīgas instrukciju kopas ir ar saviem mīnusiem un šajā gadījumā lielākais mīnuss ir ātruma zudums. Vēlāk pētījumi atklāja ka tikai 20% no instrukcijām kas ir CISC mašīnās ir atbildīgi par 80% no koda. Tas un arī fakts ka šīs kompleksās instrukcijas var tikt iestrādātas kombinācijā ar vienkāršām deva atpakaļejošu kustību pie cieši saistītājām arhitektūrām, kas tika sauktas par Reduced Instruction Set Computer (RISC).

RISC: arhitektūra ar vienkāršu cieši saistītām instrukcijām kam bieži vien vajag tikai vienu vai dažus procesora ciklus lai to izpildītu. RISC mašīnas iezīmējas ar mazu un fiksētu koda izmēru ar salīdzinoši maz instrukcijām un dažām adresēšanas metodēm. Tā rezultātā instrukciju izpilde ir ļoti ātra bet instrukciju kopums samērā vienkāršs.

CISC : arhitektūra ir sadalīta sarežģītās mikrokoda instrukcijās kas aizņem vairākus procesora ciklus lai izpildītu. Arhitektūra bieži vien ir ar lielu un mainīgu koda izmēru un piedāvā daudz spēcīgas instrukcijas un adresēšanas metodes. Salīdzinājumā ar RISC tās instrukcijas tiek izpildītas ilgāk bet ar tām ir iespējams paveikt vairāk.

Protams kad ir divas arhitektūras rodas jautājums kura no tām ir labāka. Šajā gadījumā atbilde ir atkarīga no tā kas jums ir nepieciešams. Ja risinājums bieži izmanto sarežģītas instrukcijas vai adresēšanas metodes tad visdrīzāk labākais variants ir izmantot CISC. Ja pārsvarā tiek izmantotas vienkāršas instrukcijas un adresēšanas metodes tas labāks risinājums būs RISC. Protams šīs abas izvēles arī ietekmējās no citiem faktoriem, tādiem kā cik ātrs ir procesors. Jebkurā gadījumā Jums ir jāzin kas tiek prasīts no arhitektūras lai izdarītu pareizo izvēli.

Von Neumann pret Harvard Arhitektūra

Attēlā 2.1. instrukcijas atmiņa un datu atmiņa ir attēlota kā divas dažādas. Ne vienmēr tas tā ir, gan instrukcijas, gan dati var būt arī vienā dalītā atmiņā. Patiesīb sakot tieši šī ir galvenā atšķirība starp šīm divām arhitektūrām, vai programmas un datu atmiņa ir vienuviet vai atsevišķi.

Von Neumann arhitektūra : šajā arhitektūrā programma un dati tiek glabāti kopā un ir pie tiem piekļūst izmantojot vienu ceļu. Diemžēl tas nozīmē to ka programmas un datu piekļuve var iespaidot viena otru, kas var novest pie auzkavēm.

Harvard arhitektūra : šī arhitektūra prasa lai programma un dati būtu atsevišķās atmiņās un kurām piekļūst pa dažādiem ceļiem. Rezultātā koda piekļuves nejaucas kopā ar datu piekļuvēm kas uzlabo sistēmas darbību. Kā nelielu mīnusu var minēt to ka šai arhitektūrai ir nepieciešams vairāk tehnikas, tā kā tai vajag divus ceļus un divus atmiņas čipus vai divu portu atmiņu (atmiņas čips kurš ļauj divām nesaistītam piekļuvēm vienlaicīgi).

Instrukciju kopa

Instrukciju kopa ir svarīga daļa jebkuram CPU. Tā iespaido koda izmēru, tas ir cik lielu daļu atmiņas aizņems jūsu programma. Tātad jums būtu jāizvēlas kura instrukciju kopa atbilst labāk jūsu specifiskajām vajadzībām. Iezīmes kas ir svarīgas instrukciju kopās priekš uzbūves izvēles ir:

- Instrukcijas izmērs
- Izpildes ātrums
- Pieejamie instrumenti
- Adresēšanas metodes

Instrukcijas izmērs

Instrukcija sevī iekļauj informāciju par operāciju kas būtu jāizpilda un tās operandiem. Protams mašīna ar daudz dažādām instrukcijām un adresācijas modeļiem prasīs garākus kodus kā mašīna ar dažām instrukcijām un adresācijas metodēm. Tātad CISC mašīnas ir ar garākiem kodiem nekā RISC.

Ievērojiet to ka garāki kodi ne vienmēr nozīmē to ka programma aizņems vairāk vietas kā mašīna ar īsāku kodu. Kā jau tika minēts tad CISC salīdzinājumā ar RISC ir svarīgi kas ir nepieciešams. Piemēram 10 līnijas iekš ATmega16 RISC kodā aizņem 20 baitus, katra instrukcija aizņem 16 bitus, turklāt 68030 instrukcija aizņem 4 baitus. Tātad 68030 nepārprotami uzvar. Ja tomēr ir iespējams izmantot instrukcijas kas jau piedāvātas arhitektūrā ar īsiem kodiem tad tas visdrīzāk apsteigs mašīnu ar garākiem kodiem. Mēs sakām visdrīzāk tādēļ, ka CISC mašīnas ar gariem kodiem ir tendētas aizpildīt šo deficītu ar dažāda izmēra instrukcijām. Tik ilgi kamēr pirmais baits instrukcijā skaidri norāda vai nākamos bitus vajadzēs dekodēt vai nē nav iemesla neļaut vienkāršām instrukcijām aizņemt vairāk par vienu baitu. Protams šī tehnika padara dekodēšanu daudz sarežģītaku bet tomēr tā ir labāka par lielu fiksēta uzmēra kodu. Turpretī RISC mašīnas cenšas panākt īsu bet fiksēta izmēra kodus atvieglojot instrukciju dekodēšanu.

Acīmredzami daudz vietas kodā aizņem operandi. Tātad viens veids kā samazināt instrukciju izmēru ir samazināt operandus skaitu kas ir iekodēti. Kā sekas tam var būt mūsu izveidotas četras dažādas arhitektūras balsoties uz to cik daudz operandus, tādi kā ADD pieprasa:

Steka arhitektūra: šī arhitektūra tiek saukta arī par 0-adrešu formāta arhitektūru, tai nav neviena precīzi formulēta operanda. Tā vietā operandi tiek organizēti stekos: tāda instrukcija kā ADD aizņem augstākais divas vērtības no steka, pievieno tās un novieto rezultātu stekā.

Uzkrājēja arhitektūra : šī arhitektūra tiek saukta arī par 1-adreses formāta arhitektūru, ir ar uzkrājēju kurš ir vienmēr izmantots kā viens no operandiem un kā mērķa reģistrs. Otrais operands ir specificēts skaidri.

2-adrešu fotmāta arhitektūra : šeit abi operandi ir specifiski, bet viens no tiem tiek izmantots kā mērķis lai saglabātu rezultātu. Kurš reģistrs tiek lietots šim nolūkam ir atkarīgs no procesora. ATmega16 kontrolieris izmanto pirmo reģistru kā mērķi, bet 68000 izmanto otro.

3-adrešu formāta arhitektūra : šajā arhitektūrā abi avota operandi un mērķi ir skaidri noteikti. Šī arhitektūra ir ar vis lielākajām pielaidēm, bet protams tai ir arī lielākais instrukcijas izmērs.

2.1. tabula parāda atšķirības starp arhitektūrām kad aprēķina (A+B)*C. mēs pieņemam ka ar 2 un 3 adrešu formātiem rezultāts tiek iedalīts pirmajā reģistrā. Mēs arī pieņemam ka ar 2 un 3 adrešu formāta arhitektūrām ir ielādes/novietnes arhitektūras , kur aritmētiskās instrukcijas operē tikai caur reģistriem. Pēdējā rinda tabulā norāda kur rezultāts liek saglabāts.

Izpildīšanas ātrums.

Instrukcijas izpildīšanas ātrums ir atkarīgs no vairākiem faktoriem visvairāk to iespaido arhitektūras sarežģītība, tātad uzreiz var gaidīt ka CISC mašīna patērēs vairāk ciklus instrukcijas izpildei. Tas arī ir atkarīgs no mašīnas vārda garumiem. Kā arī oscilatora frekfence definē absolūto izpildīšanas ātrumu, tā kā procesors var darboties ar 20 MHz, var atļauties izmantot divtik ciklus un tikuntā būt ātrāks kā CPU ar maksimālo frekfenci 8 MHz.

Pieejamās instrukcijas

Protams kontroliera izvēlē ir svarīgi ievērot to vai instrukcijas ir pieejamas. Instrukcijas parasti tiek iedalītas dažās klasēs:

Aritmētiski loģiskās instrukcijas: šī klase ietver sevī visas operācijas kuras kaut ko rēķina, piemēram, ADD, SUB, MUL, … un loģiskās operācijas tādas kā AND, OR, XOR, … tās arī var saturēt bitu operācijas , BSET (uzstādīt bitu), BCLR (notīrīt bitu) un BTST (pārbaudīt vai bits ir uzstādīts). Bitu operācijas ir svarīga mikrokontroliera funkcijas, tā kā tās ļauj piekļūt atsevišķiem bitiem nemainot pārējos bitus baitā. Pārvietošanas operācijas kas pārvieto reģistra datus pa vienu bitu pa labi vai pa kreisi tiek parasti piedāvāti gan loģiskajās gan aritmētiskajās operācijās. Atšķirība ir to attieksmē pret vis vērtīgāko bitu kad pārbīda pa labi (kas rezultējas kā dalīšana ar 2). Skatoties uz to aritmētiski msb (most significant bit – vis ievērojamākais bits) ir zīmes bits un to vajadzētu saglabāt kad notiek pārbīde pa labi. ja msb ir uzstādīts, tad aritmētiskā labā pārbīde saglabās msb. Kad uz to skatās ar loģisko pusi msb ir kā vēl viens bits, tātad šajā gadījuma pārbīde pa labi notīrīs msb. Jāatceras ka nav nepieciešamības saglabāt msb kad tiek veika pārbīde pa kreisi (kas rezultējas kā dalīšana ar 2). Šeit vienkāršās loģikas pārbīde tikuntā saglabās msb kamēr neizveidosies pārpilde (owerflow). Ja pārpilde notiek, tad nesaglabājot msb vienkārši ļauj rezultātam parādīties un statusa reģistrs parādīs ka rezultāts ir pārpildē.

Datu pārsūtīšana : šīs operācijas pārsūta datus starp diviem reģistriem, starp reģistriem un atmiņu, vai starp atmiņas vietām. Tie satur normālas atmiņas piekļūšanas instrukcijas, tādas kā LD (load - ielādēt) un ST (store - noglabāt), bet arī steka piekļuves operācijas PUSH un POP.

Programmas plūsma : šeit varēs atrast visas instrukcijas kas ietekmē programmas plūsmu. Tās iekļauj leciena instrukcijas kas uzstāda programmas skaitītāju uz jaunu adresi, nosacījuma nozares tāda kā BNE, subrutīnas saucieni un saucieni kas atgriežas no subrutīnām tādi kā RET un RETI.

Kontroles instrukcijas: šī klase satur visas instrukcijas kas ietekmē kontroliera operāciju, vienkāršākā no tādām instrukcijām ir NOP, kura pavēsta vai CPU kaut ko dara. Visas pārējās speciālās instruksijas, tādas kā enerģijas pārvalde, pārlāde, atkļūdošanas režīma kontrole, … arī ietilpst šajā klasē.

Adresēšanas veidi

Kad izmanto aritmētiskās instrukcijas, tad aplikācijas programmētājam ir jābūt spējīgam precizēt precīzi formulētos operandus. Operands var sastāvēt no konstantēm, reģistru datiem, vai datiem no atmiņas lokācijām. Tātad procesoram jāpiedāvā veids kā noteikt operanda tipu. Kamēr visi procesi vienmēr atļauj noteikt iepriekš minētos tipus, piekļuve pie atmiņas lokācijām var tikt paveikta ar daudz un dažādiem veidiem atkarībā no tā kas ir nepieciešams. Tātad numuri un adresēšanas veidu tipi kas tiek piedāvāti ir vēl viens no nozīmīgajiem objektiem jebkurā procesorā. Ir vairākas adresēšanas metodes, bet mēs apskatīsim tikai biežāk izmantotās.

Bezaiztures/burtiskās: šeit operands ir konstants. No aplikācijas programmētāja puses, procesors var vainu papildināt attālu instrukciju ar konstantēm (tādām kā LDI – nekavošā ielāde load immidiate – izmanto ATmega16) vai pieprasīt programmētājam lai atzīmē konstantes assambler kodā ar kādu no prefiksiem piemēram #.

Reģistrs : šeit operands ir reģistrs kas satur vērtību kuru vajadzētu izmantot lai saglabātu rezultātu.

Tiešā/absolūtā : operands ir atmiņas lokācija.

Reģistra netiešā : šeit reģistrs ir norādīts bet satur tikai atmiņas adresi no paša tiešā mērķa vai avota. Patiesā pieeja ir šīs adreses lokācijai.

Auto pieaugums : šis ir netiešas adresēšanas kur komponentes no specifiskiem reģistriem ir pievienotas vainu pirms tam vai pēc pieejas pie atmiņas lokācijas. Gadījumā ja piekļuve atmiņai notikusi pēc pievienošanas ir ļoti noderīgi izmantot iterācijas izmantojot masīvus.

Auto samazinājums : šis ir pretējais auto pieaugumam. Reģistra vērtība samazinās.

Pārvietojuma : šajā režīmā programmētājs precizē konstatēs un reģistrus, reģistra saturu pievieno konstantēm lai iegūtu galīgo atmiņas lokāciju. Tas var tikt izmantots priekš masīviem ja konstantes tiek interpretētas kā bāzes adreses un reģistri kā rādītāji masīva ietvaros.

Rādītāja : šeit divi reģistri tiek norādīti, un to saturs tiek pievienots no atmiņas adreses. Šis režīms ir līdzīgs pārvietojuma režīmam un var atkal tik izmantots ar masīviem, tos sadalot. Daži kontrolieri izmanto speciālu reģistru ar rādītāja reģistriem. Šajā gadījumā tiem nevajag būt tieši norādītiem.

Atmiņas netiešie : proframmētājs atkal nosaka reģistru, bet atmiņas lokācija tiek interpretēta kā rādītajs, tā sevī glabā pēdējās atmiņas lokāciju.

Tabula 2.2. parāda adresācijas režīmus darbībā.