**1. Ievads**

* Problēmas aktualitāte (kāpēc stekatmiņas analīze ir svarīga AVR mikrokontrolieros)
* Darba mērķis un uzdevumi
* Pētījuma metodoloģija
* Darba struktūras apraksts

**2. Teorētiskais pamatojums**

* AVR mikrokontrolieru arhitektūras apskats
  + Atmiņas organizācija AVR mikrokontrolieros
  + RAM, Flash un EEPROM raksturojums
* Stekatmiņas koncepcija un tās nozīme
  + Steka darbības principi
  + Funkciju izsaukumu mehānisms
  + Lokālo mainīgo glabāšana
  + AVR vispārīgie reģistri un operandu apzīmējumi (Rd, Rr)
* Stekatmiņas pārplūdes problēmas un to sekas

**3. Esošo risinājumu analīze**

* GCC steka lietojuma analīzes rīki (-fstack-usage)
* AVR-GCC kompilatora risinājumi
* optimizācija
* Ierobežojumi esošajos risinājumos
  + Rekursīvo funkciju apstrāde
  + Netiešo funkciju izsaukumu problēmas

**4. AVR stekatmiņas izmantošanas analizatora izstrāde**

* Izstrādes prasības un mērķi
* Izmantotās tehnoloģijas un rīki
* Analizatora arhitektūras projektējums
* Galvenie algoritmi
  + Statiskie steka izmantošanas aprēķini
  + Rekursīvo funkciju analīze
  + Izsaukumu grafika rekonstrukcija no asemblera
  + Maksimālā steka dziļuma aprēķināšana

**5. Analizatora implementācija**

* Avota koda kompilēšana
* Asemblera koda analīze
* Rekursīvo funkciju noteikšana
* Izsaukumu grafika veidošana
* Steka izmantojuma aprēķināšana
* Rezultātu vizualizācija un atskaišu ģenerēšana

**2. Teorētiskais pamatojums**

* **AVR mikrokontrolieru arhitektūras apskats**

Centrālais elements AVR mikrokontrolieros ir 8 bitu mikroprocesora kodols jeb centrālā procesora ierīce (CPU), kas balstīts uz RISC (Reduced Instruction Set Computer) arhitektūras principiem. Šī kodola pamatā atrodas aritmētiski loģiskā ierīce (ALU), kas nodrošina instrukciju izpildi. Saskaņā ar sistēmas takts signālu, izmantojot komandu skaitītāja (Program Counter – PC) saturu, no programmu atmiņas tiek izgūta nākamā izpildāmā instrukcija. Vienlaikus ar instrukcijas izgūšanu notiek iepriekšējās instrukcijas izpilde, kas ļauj sasniegt veiktspēju līdz 1 MIPS uz katru MHz takts frekvences.

ALU ir tieši savienots ar vispārējās nozīmes reģistriem (General Purpose Registers – GPR), kuru kopējais skaits ir 32. Katrs no šiem reģistriem ir viena baita (astoņu bitu) lielumā. GPR atrodas operatīvās atmiņas adresējamās telpas sākumā, taču fiziski tie nav tās sastāvdaļa. Tas ļauj šiem reģistriem piekļūt gan atmiņas elementiem, gan reģistriem, kas ir viena no AVR arhitektūras īpatnībām, kas būtiski uzlabo mikrokontroliera darbības efektivitāti un veiktspēju.

Galvenā atšķirība starp reģistriem un operatīvo atmiņu ir tajā, ka ar reģistriem iespējams veikt plašu operāciju klāstu, tostarp aritmētiskās, loģiskās un bitu manipulācijas, savukārt operatīvajā atmiņā var tikai ierakstīt datus no reģistriem. [1]

* **Atmiņas organizācija AVR mikrokontrolieros**

AVR mikrokontrolieros tiek izmantota Hārvardas arhitektūra, kuras būtiska īpatnība ir programmu un datu atmiņu fiziska un loģiska nošķiršana. Tas nozīmē, ka mikrokontrolierī ir atdalīti gan programmatūras un datu atmiņu adresēšanas apgabali, gan piekļuves kopnes. Šāda arhitektūra ļauj vienlaikus izgūt instrukciju un apstrādāt datus, tādējādi uzlabojot izpildes efektivitāti.

Datu atmiņa mikrokontrolieros parasti sastāv no divām neatkarīgām komponentēm – operatīvās atmiņas (SRAM) un pastāvīgās datu atmiņas (EEPROM), kuras katra atrodas savā adresējamā telpā. Savukārt instrukciju jeb programmu atmiņa tiek realizēta ar Flash tipa ROM.

**Programmu atmiņa (Flash ROM)**

Programmu atmiņa (Flash ROM) ir paredzēta mikrokontroliera izpildāmo instrukciju glabāšanai. Tā ir 16 bitu organizācijas atmiņa, kuras ietilpība dažādos AVR mikrokontrolieru modeļos var svārstīties no 1 līdz 256 KB. Flash tehnoloģijas galvenā priekšrocība ir elektriskās pārrakstīšanas iespēja, kas ļauj vairākkārtīgi ierakstīt un dzēst informāciju.

Programmas ierakstīšana Flash atmiņā ir iespējama gan ar ārējiem programmatoriem, gan caur SPI interfeisu. Vairums AVR mikrokontrolieru, izņemot Tiny11 un Tiny28, atbalsta tiešsaistes programmēšanu (In-System Programming, ISP), kas ļauj ierakstīt programmu tieši uz samontētas shēmas plates.

Turklāt Mega sērijas mikrokontrolierī atbalsta arī pašprogramēšanu (self-programming), tas dod tiem iespēju pašiem mainīt savu programmu atmiņas saturu izpildes laikā. Šī funkcionalitāte ļauj izstrādāt adaptīvas un elastīgas sistēmas, kas var mainīt savu darbības loģiku atkarībā no iekšējiem vai ārējiem nosacījumiem. [1]

Saskaņā ar Atmel korporācijas tehnisko dokumentāciju, AVR otrās paaudzes mikrokontrolieru Flash atmiņai tiek garantēts vismaz 10 000 pārrakstīšanas ciklu, bet tipiskos apstākļos šis skaits var sasniegt līdz pat 100 000 ciklu. [2]

* **RAM, Flash un EEPROM raksturojums**

**Datu atmiņa**

AVR mikrokontrolieros datu atmiņa ir loģiski sadalīta trīs daļās: reģistru atmiņā, operatīvajā atmiņā (RAM) un neatkarīgajā datu atmiņā (EEPROM). Katra no šīm komponentēm pilda noteiktu funkciju mikrokontroliera darbībā.

**Reģistru atmiņa (GPR un I/O reģistri)**

Reģistru atmiņu veido 32 vispārējās nozīmes reģistri (GPR – General Purpose Registers), kas organizēti kā vienots reģistru fails, kā arī speciālie ievades/izvades reģistri (I/O reģistri). Lai gan šie reģistri ir adresējami tajā pašā adresēšanas telpā kā operatīvā atmiņa, fiziski tie nav tās sastāvdaļa.

Ievades/izvades reģistru zonā atrodas dažādi kontroles un stāvokļa reģistri, kas nosaka mikrokontroliera darbību. Tie ietver gan sistēmas vadības reģistrus (piemēram, statusa reģistrus), gan perifērijas ierīču kontroles reģistrus. Patiesībā mikrokontroliera vadība būtībā ir šo reģistru pārvaldība, jo tie ir atbildīgi par visa sistēmas funkcionālā aprīkojuma konfigurēšanu un darbību.

**Neatkarīgā datu atmiņa (EEPROM)**

Datu ilgtermiņa glabāšanai tiek izmantota EEPROM (Electrically Erasable Programmable Read-Only Memory) – elektriski izdzēšama un pārrakstāma pastāvīgā atmiņa. Tā ir noderīga informācijai, kas var mainīties darbības laikā, piemēram, lietotāja iestatījumiem, konfigurācijas parametriem, sērijas numuriem, kriptogrāfiskajiem atslēgām u.c.

Visi AVR mikrokontrolieri ir aprīkoti ar EEPROM moduli, kura izmērs atkarībā no modeļa var būt no 64 baitiem līdz 4 KB. EEPROM saturs ir pieejams mikrokontroliera programmai izpildes laikā. Ierakstīšanu un nolasīšanu iespējams veikt gan programmēšanas laikā (ar ārēju programmatoru vai SPI interfeisu), gan arī darbības laikā no paša mikrokontroliera koda. EEPROM tipiski garantē vismaz 100 000 pārrakstīšanas ciklu, saglabājot datus arī pēc barošanas atslēgšanas.

**Operatīvā atmiņa (SRAM)**

Operatīvā atmiņa (RAM – Random Access Memory) AVR mikrokontrolieros tiek realizēta kā statiskā RAM (SRAM), kurai ir baitu organizācija. Tā tiek izmantota mainīgo, kaudzes (stack) un citu īslaicīgu datu glabāšanai izpildes laikā. Atmiņas apjoms dažādos mikrokontrolieru modeļos var būt no 64 baitiem līdz 4 KB.

Tā kā SRAM ir gaistoša atmiņa, tās saturs tiek zaudēts, tiklīdz tiek pārtraukta barošana. Tomēr datu nolasīšanas un ierakstīšanas ciklu skaits nav tehniski ierobežots. Dažiem mikrokontrolieriem ir iespējams pievienot arī ārējo statisko RAM līdz 64 KB, kas ievērojami paplašina iespējas lielāku datu apstrādei. [1]

[1] <https://myrobot.ru/stepbystep/mc_architecture.php>

[2] <https://ww1.microchip.com/downloads/en/DeviceDoc/en590320.pdf>

* **Stekatmiņas koncepcija un tās nozīme**
* **Steka darbības principi**

Steks (stack – grēda, krāvums) ir datu struktūra, kura darbojas pēc principa "pēdējais iekšā, pirmais ārā" (LIFO – Last In First Out). Steku raksturo divas pamatoperācijas: ievietošana (push) un izņemšana (pop). Ievietošanas operācija pievieno jaunu elementu steka augšpusē. Izņemšanas operācija izņem elementu, kurš atrodas steka augšpusē, un atgriež to lietotājam. Apskate (peek) – steka elementa pārbaude bez tā izņemšanas. [3]

AVR mikrokontrolieros steka rādītājs (SP- Stack Pointer ) ir 16-bitu reģistrs, kas norāda uz pēdējo aizņemto adresi stekā. Steka rādītāja inicializācijas vērtība ir iestatīta uz augstāko SRAM adresi.

* **Funkciju izsaukumu mehānisms**

**Funkciju izsaukums: CALL / RCALL**

Funkciju izsaukšanas laikā tiek izmantotas instrukcijas CALL (vai RCALL, ja relatīvs adresējums). Funkcijas izsaukšanas laikā notiek šādas darbības:

1. Programmas skaitītāja (PC) vērtība, kas norāda uz nākamo instrukciju pēc funkcijas izsaukuma, tiek ievietota stekā.
   * AVR mikrokontrolieriem ar līdz 128 KB programmatūras atmiņu, PC ir 16 biti vai 22 biti (atkarībā no ierīces), tāpēc tiek saglabāti 2 vai 3 baiti.
2. SP tiek samazināts (jo steks aug uz zemākām adresēm).
3. Vadība tiek nodota uz funkcijas sākuma adresi (CALL mērķis). [4]

**Atgriešanās no funkcijas: RET**

Instrukcija RET tiek izmantota, lai atgrieztos no funkcijas:

1. No steka tiek izņemta iepriekš saglabātā PC vērtība (2 vai 3 baiti).
2. SP tiek palielināts, atbrīvojot iepriekš izmantotās steka šūnas.
3. Vadība tiek nodota uz šo adresi – t.i., tiek atgriezta kontrole funkciju izsaukušajai vietai. [4]

Lai nodrošinātu funkciju darbības izolāciju un izvairītos no datu zuduma, AVR arhitektūrā tiek plaši izmantotas instrukcijas PUSH un POP. Tās ļauj īslaicīgi saglabāt reģistru vērtības steka funkcijas izpildes laikā un pēc tam tās atjaunot, nodrošinot konteksta saglabāšanu.

**PUSH – Reģistra ievietošana stekā**

Instrukcija PUSH Rr veic sekojošas darbības:

1. SP samazinās par 1, norādot uz nākamo brīvo vietu stekā.
2. Reģistra Rr saturs tiek ierakstīts SRAM adresē, uz kuru tagad norāda SP [4]

**POP – Reģistra atjaunošana no steka**

Instrukcija POP Rd darbojas šādi:

1. No SRAM adreses, uz kuru norāda SP, tiek nolasīta vērtība.
2. Šī vērtība tiek ievietota reģistrā Rd.
3. SP palielinās par 1, jo šī steka šūna vairs nav aizņemta. [4]

* **Lokālo mainīgo glabāšana**

AVR mikrokontrolieros lokālo mainīgo glabāšanai tiek izmantots steks, kas ir kritiski svarīga atmiņas pārvaldības mehānisma sastāvdaļa. Atšķirībā no globālajiem mainīgajiem, kuri tiek glabāti fiksētās SRAM adresēs, lokālie mainīgie tiek izvietoti stekā un ir pieejami tikai funkcijas izpildes laikā. Sarežģītāki datu tipi, piemēram, masīvi un struktūras, stekā tiek glabāti secīgi. [5]

**Lokālo mainīgo piekļuves mehānisms**

AVR procesora instrukciju komplektā nav tiešu instrukciju darbam ar steku, izņemot PUSH un POP operācijas. Tādēļ, lai piekļūtu lokālajiem mainīgajiem, tiek izmantots netiešās adresācijas mehānisms, galvenokārt ar Y reģistra (r28) [4]

Lokālo mainīgo piekļuves process:

1. Funkcijas sākumā tiek saglabāti iepriekšējie Y reģistra saturs (ja tas tiek izmantots kā ietvara rādītājs)
2. Y reģistrs tiek iestatīts, lai norādītu uz steka ietvara sākumu
3. Piekļuve lokālajiem mainīgajiem notiek ar nobīdi attiecībā pret Y reģistru

* **AVR vispārīgie reģistri un operandu apzīmējumi (Rd, Rr)**

AVR arhitektūrā ir pieejami 32 vispārīgie 8 bitu reģistri, kas apzīmēti kā R0 līdz R31. Šie reģistri ir tieši pieejami instrukcijām un atrodas procesora vispārīgajā reģistru failā. Tie tiek izmantoti gan aritmētiskajām un loģiskajām operācijām, gan datu pārnesei un uzglabāšanai.

**Reģistru priekšrocības:**

* Tie ir pieejami vienā ciklā (viena takts laikā), tādēļ darbības ar tiem ir ļoti ātras.
* Reģistri R26–R31 var tikt apvienoti pāros (X, Y, Z) un izmantoti kā indeksētie rādītāji (LD, ST u.c. instrukcijās).

**Reģistru apzīmējumi instrukcijās: Rd un Rr**

Instrukciju sintaksē bieži tiek izmantoti saīsinājumi:

* Rd (Destination Register- mērķa reģistrs) – reģistrs, kurā tiks ievietots rezultāts.
* Rr (Source Register - avota reģistrs) – reģistrs, no kura tiek ņemta ievades vērtība (piemēram, saskaitāmais).

Šie apzīmējumi netiek lietoti programmā tieši, bet tiek izmantoti instrukciju formātos un dokumentācijā, lai apzīmētu lomu, kāda katram reģistram ir konkrētajā instrukcijā.

**Piemērs (asemblera valodā) – Aritmētiska operācija:**

ADD R16, R17 ; R16 = R16 + R17

R16 ir Rd – rezultāts tiek glabāts šeit,

R17 ir Rr – tas tiek pieskaitīts.

**Ierobežojumi dažām instrukcijām**

Ne visas instrukcijas ļauj izmantot jebkuru no 32 reģistriem:

* Instrukcija LDI (Load Immediate) darbojas tikai ar reģistriem R16 līdz R31, jo tai nepieciešams vairāk bitu, lai kodētu gan konstanti, gan reģistra adresi.
* Instrukcijas, kas izmanto rādītājus (LD, ST, LPM, utt.), izmanto reģistru pārus (R26:R27, R28:R29, R30:R31) kā adresācijas reģistrus. [4]

|  |  |  |
| --- | --- | --- |
| **Apzīmējums** | **Nozīme** | **Apraksts** |
| R0–R31 | Vispārīgie reģistri | Pieejami operācijām, glabāšanai, rādītājiem |
| Rd | Destination Register | Reģistrs, kurā nonāk rezultāts |
| Rr | Source Register | Reģistrs, no kura ņem ievades vērtību |
| LDI | Load Immediate | Atļauts tikai R16–R31 |
| PUSH/POP | Steka operācijas | Var izmantot R0–R31 |

[3] <https://www.cprogramming.com/tutorial/computersciencetheory/stack.html>

[4] <https://ww1.microchip.com/downloads/en/DeviceDoc/AVR-InstructionSet-Manual-DS40002198.pdf>

[5]<https://www.bogotobogo.com/cplusplus/files/embed/OReilly_Programming_Embedded_Systems_Second_edition_ebook.pdf>

* **Stekatmiņas pārplūdes problēmas un to sekas**

Kad programma mēģina izmantot vairāk stekatmiņas, nekā fiziski ir pieejams – to sauc par stekatmiņas pārplūdi. Tajā brīdī steka rādītāja nonāk aiz piekļaujamas zonas, izraisot nepareizo programmas izpildi vai pat pilnīgo sistēmas avāriju. AVR mikrokontrolieros šis ir īpaši kritisks jautājums šādu iemeslu dēļ:

1. **Neparedzama uzvedība**: Steka pārplūde var izraisīt globālo mainīgo pārrakstīšanu, kas noved pie datu bojāšanas un neparedzamas programmas uzvedības.
2. **Sistēmas avārijas**: Steka pārplūde var izraisīt pilnīgu sistēmas avāriju, prasot manuālu restartēšanu.
3. **Ierobežotie atmiņas resursi**: AVR mikrokontrolieriem ir ļoti maz SRAM (piemēram, ATmega328P ir tikai 2KB), no kuras steks ir tikai daļa.
4. **Nav atmiņas aizsardzības mehānismu**: Atšķirībā no modernu operētājsistēmu arhitektūrām, AVR mikrokontrolieriem nav atmiņas aizsardzības bloku vai virtuālās atmiņas, kas varētu konstatēt vai novērst pārplūdes.
5. **Grūti atkļūdojamas problēmas**: Stekatmiņas pārplūdes bieži ir grūti identificējamas un atkļūdojamas, jo to simptomi var parādīties tālu no patiesas cēloņa un neparādīties vienmēr.

**3. Esošo risinājumu analīze**

* **GCC steka lietojuma analīzes rīki (-fstack-usage)**
* **AVR-GCC kompilatora risinājumi**
* **Optimizācija**
* **Ierobežojumi esošajos risinājumos** 
  + **Rekursīvo funkciju apstrāde**
  + **Netiešo funkciju izsaukumu problēmas**

**4. AVR stekatmiņas izmantošanas analizatora izstrāde**

* **Izstrādes prasības un mērķi**
* **Izmantotās tehnoloģijas un rīki**

Autors izstrādājot steka analizatoru izmantoja python un asamblera programmēšanas valodas, kā arī avr-gcc, avr-objdump un avr-size rīkus statiskas analīzes daļai un simavr un gdb rīkus dinamiskai analīzei.

* **Analizatora arhitektūras projektējums**
* **Galvenie algoritmi** 
  + **Statiskie steka izmantošanas aprēķini**
  + **Rekursīvo funkciju analīze**
  + **Izsaukumu grafika rekonstrukcija no asemblera**
  + **Maksimālā steka dziļuma aprēķināšana**