**VENTSPILS AUGSTSKOLA**

**INFORMĀCIJAS TEHNOLOĢIJU FAKULTĀTE**

**BAKALAURA DARBS**

**AVR mikrokontrolieru stekatmiņas izmantošanas analizatora izstrāde**

Autors Ventspils Augstskolas

Informācijas tehnoloģiju fakultātes

bakalaura studiju programmas „Datorzinātnes”

3. kursa students

Anatolijs Koļesņevs

Matr.nr. 22020011

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(paraksts)

Fakultātes dekāns doc. Dr.sc.comp. Vairis Caune

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(paraksts)

Zinātniskais vadītājs \_\_\_\_\_\_\_\_\_\_\_ Mg. sc. ing. Jānis Šmēdiņš \_\_\_\_\_\_\_\_\_\_\_

(ieņemamais amats, zinātniskais nosaukums, vārds, uzvārds)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(paraksts)

1. **Ievads**
   1. **Problēmas aktualitāte (kāpēc stekatmiņas analīze ir svarīga AVR mikrokontrolieros)**
   2. **Darba mērķis un uzdevumi**
   3. **Pētījuma metodoloģija**
   4. **Darba struktūras apraksts**
2. **Teorētiskais pamatojums**
   1. **AVR mikrokontrolieru arhitektūras apskats**

Katra AVR mikrokontroliera "sirds" ir 8 bitu centrālais procesors, kas balstīts uz RISC (Reduced Instruction Set Computer) principiem. RISC pieeja nozīmē vienkāršākas un efektīvākas instrukcijas, kas ļauj mikroprocesoram darboties ātrāk ar mazāku elektrības patēriņu.

AVR procesora darbības pamatā ir aritmētiski loģiskā ierīce (ALU), kas izpilda visas matemātiskās un loģiskās operācijas. Šī sistēma ir izstrādāta tā, lai vienlaicīgi notiktu divas darbības – kamēr ALU izpilda pašreizējo instrukciju, programmskaitītājs (PC) jau iegūst nākamo instrukciju no programmu atmiņas. Šāda "cauruļvada" pieeja ļauj AVR mikroprocesoram sasniegt iespaidīgu veiktspēju – vienu miljonu instrukciju sekundē (MIPS) uz katru takts frekvences MHz.

Īpaši interesanta AVR arhitektūras iezīme ir tās reģistru organizācija. Procesorā ir 32 vispārējās nozīmes reģistri (GPR), katrs ir viena baita (astoņu bitu) lielumā. Šie reģistri atrodas operatīvās atmiņas adresējamās telpas sākumā, bet tie nav fiziski operatīvās atmiņas daļas. 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. [1]

Būtiska atšķirība starp reģistriem un operatīvo atmiņu ir to funkcionalitāte – reģistri spēj veikt daudzveidīgas operācijas (saskaitīšana, atņemšana, loģiskās operācijas, bitu pārbīdes utt.), kamēr operatīvajā atmiņā var tikai glabāt datus. Tāpēc efektīvās mikrokontroliera programmās cenšas pēc iespējas vairāk izmantot reģistrus, un tikai nepieciešamības gadījumā dati tiek pārvietoti uz operatīvo atmiņu.

* + 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)** 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. [2]

* + 1. **RAM, Flash un EEPROM raksturojums**

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 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.

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.

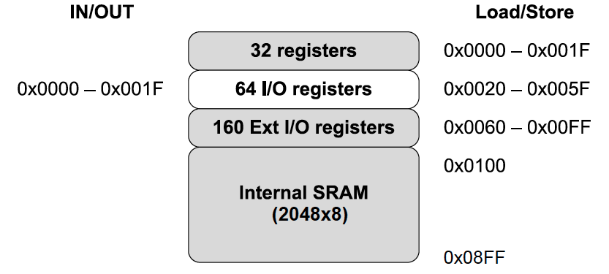
**Operatīvā atmiņa** (RAM – Random Access Memory) AVR mikrokontrolieros tiek realizēta kā statiskā RAM (SRAM), kurai ir baitu organizācija. Tā tiek izmantota mainīgo, kaudzes (stack) un citu īslaicīgu datu glabāšanai izpildes laikā. Atmiņas apjoms dažādos mikrokontrolieru modeļos var būt no 64 baitiem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

**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.



*2.1. att.* AVR mikrokontroliera SRAM datu atmiņa [7]

2.1. attēls attēlo AVR mikrokontroliera atmiņas karti un adresēšanas struktūru. Tas ir būtisks, jo parāda, kā AVR arhitektūrā ir organizēta adrešu telpa, iekļaujot reģistrus un SRAM. Attēls parāda četras galvenās komponentes, kas katra aizņem noteiktu adrešu diapazonu.

**32 reģistri** (0x0000 – 0x001F) - Šie ir mikrokontroliera pamatreģistri, kas tiek izmantoti tiešām datu operācijām. Tie ir pieejami gan izmantojot IN/OUT instrukcijas, gan ar Load/Store instrukcijām.

**64 I/O reģistri** (0x0020 – 0x005F) - Šie reģistri tiek izmantoti, lai kontrolētu dažādus mikrokontroliera perifērijas ierīces un funkcijas, piemēram, portus, taimeri, ADC utt.

**160 paplašinātie I/O reģistri** (0x0060 – 0x00FF) - Šie papildu I/O reģistri nodrošina piekļuvi vēl plašākam mikrokontroliera funkciju klāstam.

**Iekšējā SRAM** (0x0100 – 0x08FF) - Šī ir mikrokontroliera operatīvā atmiņa ar izmēru 2048x8 biti (2KB). Šeit glabājas programmas dati, mainīgie, un steks. Steks tipiski sākas no augšējās atmiņas adreses (0x08FF) un "aug" lejup virzienā uz zemākām adresēm.

Attēls ir īpaši vērtīgs, jo parāda, ka AVR arhitektūrā ir divas dažādas metodes, kā piekļūt reģistriem - caur IN/OUT instrukcijām un caur Load/Store instrukcijām, kas lieto atšķirīgas adreses. Šī duālā adresēšanas sistēma ir AVR arhitektūras raksturīga iezīme, kas programmētājiem jāņem vērā, rakstot efektīvu asemblera kodu.

**Funkciju izsaukums: CALL / RCALL**

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

1. Programmaskaitītāja (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 vietai, kur funkcija tika izsaukta. [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 funkcijas izpildes laikā un pēc tam tās atjaunot, nodrošinot konteksta saglabāšanu.

**PUSH – Reģistra ievietošana stekā**

Instrukcija PUSH Rr veic sekojošas darbības:

1. Reģistra Rr saturs tiek ierakstīts SRAM adresē, uz kuru tagad norāda SP
2. SP samazinās par 1, norādot uz nākamo brīvo vietu stekā. [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]
   1. **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, pp. 100 – 104]

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 reģistru (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
   1. **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.

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]

2.1. tabula

**AVR mikrokontroliera reģistru un instrukciju pārskats**

|  |  |  |
| --- | --- | --- |
| **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 |

2.1. tabulā ir iekļauta informācija par vispārīgajiem reģistriem (R0-R31), kurus var izmantot dažādām operācijām, datu glabāšanai un kā adrešu rādītājus. Papildus tam ir aprakstīti speciālie reģistru apzīmējumi, piemēram, mērķa reģistrs (Rd) un avota reģistrs (Rr), kas tiek izmantoti instrukciju pierakstā.

* 1. **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.

**Neparedzama uzvedība**: Viena no nopietnākajām stekatmiņas pārplūdes sekām ir neparedzama programmas uzvedība. Kad steks pārplūst, tas sāk pārrakstīt atmiņas apgabalus, kas sākotnēji tam nebija paredzēti. Šis process izpaužas kā šķietami neizskaidrojamas mainīgo vērtību izmaiņas programmā. Piemēram, sensors, kas iepriekš rādīja korektas vērtības, pēkšņi sāk uzrādīt kļūdainas lasījumus, vai LED indikatori sāk mirgot neparedzētā secībā. Šādas uzvedības atkļūdošana ir ārkārtīgi sarežģīta, jo problēmas izpausmes var būt ļoti attālinātas no to patiesā cēloņa.

**Sistēmas avārijas**: Stekatmiņas pārplūde bieži izraisa pilnīgu sistēmas avāriju, kad mikrokontrolieris pārstāj reaģēt uz ievadi vai izpildīt programmu. Tas notiek tāpēc, ka steka pārplūde var bojāt atgriešanās adreses, kas nepieciešamas funkciju izsaukumu pareizai izpildei. Ja atgriešanās adrese tiek bojāta, programmas izpildes plūsma var nonākt neparedzētā koda segmentā vai adresē, kas nav derīga izpildei.

**Ierobežotie atmiņas resursi**: AVR mikrokontrolieri ir projektēti ar ļoti ierobežotiem atmiņas resursiem. Piemēram, plaši izmantotajam ATmega328P mikrokontrolierim, kas ir Arduino Uno platformas pamatā, ir tikai 2KB SRAM atmiņas. No šiem 2048 baitiem stekatmiņa ir tikai daļa, jo atmiņa tiek dalīta starp globālajiem mainīgiem, dinamisko atmiņu un steku.

Šo ierobežoto resursu kontekstā stekatmiņas pārplūde kļūst par īpaši aktuālu problēmu. Ja programma satur dziļi iegultas funkciju izsaukumu hierarhijas vai lielu daudzumu lokālo mainīgo, 2KB SRAM var ātri izrādīties nepietiekami.

**Nav atmiņas aizsardzības mehānismu**: Atšķirībā no modernām datoru operētājsistēmām, kur pastāv atmiņas aizsardzības mehānismi, AVR mikrokontrolieri darbojas bez šādām aizsardzības sistēmām. Tas nozīmē, ka pārplūdušais steks var bez jebkādiem ierobežojumiem pārrakstīt jebkuru atmiņas apgabalu, un aparatūra par to neziņos un nemēģinās novērst.

**Grūti atkļūdojamas problēmas**: Stekatmiņas pārplūdes radītās problēmas ir ārkārtīgi grūti atkļūdot vairāku iemeslu dēļ:

Pirmkārt, simptomi var parādīties tālu no patiesa cēloņa. Piemēram, funkcija, kas izraisa steka pārplūdi, var pabeigt savu izpildi bez acīmredzamām problēmām, bet tās radītie bojājumi parādīsies tikai vēlāk programmas darbībā, kad cita funkcija mēģinās piekļūt bojātajiem datiem.

Otrkārt, problēmas var neparādīties konsekventi. Tās var būt atkarīgas no konkrētiem izpildes apstākļiem vai ievades datiem, kas padara atkļūdošanu vēl sarežģītāku. Piemēram, steka pārplūde var notikt tikai tad, kad lietotājs ievada īpaši garu tekstu vai kad vairākas funkcijas tiek izsauktas specifiskā secībā.

Treškārt, tradicionālās atkļūdošanas metodes, piemēram, izdrukāšana (debug print), var nedarboties, jo pati atkļūdošanas funkcija var patērēt papildu stekatmiņu, tādējādi pasliktinot problēmu vai mainot tās izpausmi. Šo efektu dažkārt dēvē par "Heizenberga nenoteiktību atkļūdošanā" — mēģinājums novērot problēmu maina pašu problēmu. [6, pp. 89-183]

Ņemot vērā stekatmiņas pārplūdes nopietno raksturu un potenciālo ietekmi, iegulto sistēmu projektēšanā ir nepieciešams ievērot īpašu piesardzību. Tas ir īpaši svarīgi kritiskās lietojumprogrammās, piemēram:

1. Medicīnas ierīcēs, kur programmatūras atteice var apdraudēt pacientu dzīvību.
2. Automobiļu vadības sistēmās, kur atteice var izraisīt avāriju.
3. Industriālās vadības sistēmās, kur atteice var izraisīt ražošanas zaudējumus vai pat bojāt aprīkojumu.

Stekatmiņas pārplūdes novēršana ir galvenais uzdevums šo sistēmu drošības un uzticamības nodrošināšanā. Efektīva stekatmiņas pārvaldība ietver gan programmatūras izstrādes paņēmienus, gan aparatūras resursu plānošanu, gan testēšanas un verifikācijas metodoloģijas.

1. **Esošo risinājumu analīze**
   1. **GCC steka lietojuma analīzes rīki (-fstack-usage)**
   2. **AVR-GCC kompilatora risinājumi**
   3. **Optimizācija**
   4. **Ierobežojumi esošajos risinājumos** 
      1. **Rekursīvo funkciju apstrāde**
      2. **Netiešo funkciju izsaukumu problēmas**
2. **AVR stekatmiņas izmantošanas analizatora izstrāde**
   1. **Izstrādes prasības un mērķi**

Galvenās prasības un mērķi izstrādājot AVR stekatmiņas analizatoru ir šādi:

1. **Precīza steka izmantojuma aprēķināšana** - nodrošināt rīku, kas var ticami prognozēt maksimālo steka izmantojumu mikrokontroliera programmai, kura ir uzrakstīta C valodā.
2. **Ērti izmantojama** - nodrošināt saprotamu un viegli izmantojamu lietotāja saskarni.
3. **Statiskā analīze bez programmas izpildes** - veikt analīzi, balstoties tikai uz pirmkodu un kompilētu asemblera kodu, bez nepieciešamības programmu reāli izpildīt.
4. **Rekursīvo funkciju atbalsts** - korekti analizēt un aprēķināt steka izmantojumu programmās ar rekursīvām funkcijām, ņemot vērā to maksimālo dziļumu.
5. **Optimizācijas atbalsts** - nodrošināt izvēli starp nepieciešamo optimizācijas līmeni.
6. **Detalizēta funkciju analīze** - nodrošināt informāciju par katras funkcijas individuālo steka izmantojumu un tās ietekmi uz kopējo steku.
7. **Izsaukumu grafika vizualizācija** - radīt vizuālu programmas funkciju izsaukumu struktūras attēlojumu ar īpašu uzsvaru uz rekursīvajām funkcijām.
8. **Atbalsts dažādiem AVR mikrokontrolieriem** - nodrošināt analizatora darbību ar dažādiem AVR sērijas mikrokontrolieriem, ņemot vērā to atmiņas īpatnības.
9. **Drošības rezerves iestrāde** - iekļaut aprēķinos drošības koeficientu, lai kompensētu neparedzētus steka izmantojuma gadījumus.
10. **Daudzvalodību atbalsts** - nodrošināt lietotāja saskarni gan latviešu, gan angļu valodā.
    1. **Izmantotās tehnoloģijas un rīki**

Autors izstrādājot steka analizatoru kā galveno programmēšanas valodu izvēlējas Python tās universāluma, vienkāršības un plašo bibliotēku dēļ. Python nodrošina:

* Ērtu teksta apstrādi un regulāro izteiksmju lietošanu.
* Vienkāršu integrāciju ar ārējiem rīkiem, izmantojot subprocess moduli.
* Efektīvu datu struktūru apstrādi grafu un algoritmu realizācijai.

Lai veiktu nepieciešamas kompilācijas tika izmantoti AVR GCC kompilatora komplekta rīki:

* avr-gcc - C koda kompilēšanai uz AVR asemblera kodu.
* avr-objdump - ELF failu disasemblēšanai, lai iegūtu asemblera kodu analīzei.
* avr-size - programmas atmiņas sekciju izmēru noteikšanai.

Statiskās analīzes procesā tiek izmantotas regulārās izteiksmes asemblera koda parsēšanai. Analizators izvelk un apstrādā GCC kompilatora ģenerētos steka izmantojuma (.su) failus apvienojot GCC sniegtos datus ar paša aprēķināto steka lietojumu, izmantojot konservatīvo pieeju - tiek izmantota lielākā no abām vērtībām, lai nodrošinātu drošu novērtējumu.

Analizators izmanto grafu teorijas algoritmus, lai analizētu funkciju izsaukumu struktūru un noteiktu “sliktāko ceļu”.

**Dinamiskās analīzes rīki:**

Lai validētu statiskās analīzes rezultātus, izmantotas dinamiskās analīzes metodes:

* **Simavr** - AVR mikrokontrolieru simulators.
* **GDB** - atkļūdotājs steka izmantojuma izsekošanai reālā laikā.

**Vizualizācijas rīki:**

Izsaukumu grafika vizualizācijai izmantots:

* DOT formāts grafiku specificēšanai.
* Graphviz pakotne DOT failu pārvēršanai attēlos.
  1. **Analizatora arhitektūras projektējums**

Analizatora arhitektūra veidota modulāri, ar skaidri nodalītiem atbildības līmeņiem:

1. Ievades apstrādes modulis:

* C pirmkoda parsēšana.
* Avota koda analīze rekursīvo funkciju meklēšanai.
* Kompilatora karogu apstrāde un kompilēšanas procesa vadība.

1. Kompilēšanas modulis:

* C koda kompilēšana uz ELF formātu.
* Kompilatora steka izmantojuma atskaišu (.su faili) savākšana.
* ELF faila disasemblēšana asemblera koda iegūšanai.

1. Asemblera koda analīzes modulis:

* Asemblera koda parsēšana pa funkcijām.
* Steka operāciju (push, pop, call, sbiw, adiw) atpazīšana un analīze.
* Funkciju izsaukumu grafika rekonstrukcija.

1. Rekursijas analīzes modulis:

* Rekursīvo funkciju identificēšana.
* Rekursijas dziļuma ierobežojumu noteikšana.
* Rekursīvo funkciju steka izmantojuma aprēķināšana.

1. Steka kalkulācijas modulis:

* Individuālo funkciju steka izmantojuma aprēķins.
* Maksimālā kopējā steka dziļuma aprēķins.
* Drošības rezerves piemērošana.

1. Vizualizācijas un atskaišu modulis:

* Teksta atskaišu ģenerēšana.
* JSON formāta atskaites.
* Izsaukumu grafika DOT formāta ģenerēšana.

1. Lietotāja saskarnes modulis:

* Komandrindiņas argumentu apstrāde.
* Dažādu valodu atbalsts.
* Rezultātu formatēšana un attēlošana.
  1. **AVR mikrokontrolieru steka izmantojuma analīzes metodes un algoritmi** 
     1. **Statiskie steka izmantošanas aprēķini**

Statiskās analīzes metodes ļauj noteikt steka izmantojumu, analizējot programmas kodu bez tās izpildes. AVR steka analizators izmanto divas metodes: GCC steka lietojuma pārskatu analīzi un asamblera koda analīzi.

Rīks izmanto GCC kompilatora -fstack-usage opciju, kas ģenerē .su failus ar informāciju par katras funkcijas steka izmantojumu. Šī metode nodrošina precīzu steka izmantojuma informāciju par katru individuālo funkciju, ko aprēķinājis kompilators. Taču tā neņem vērā funkciju izsaukumu ķēdes vai rekursiju, kas var izraisīt steka pārplūdi.

Papildus GCC pārskatiem, rīks disasemblē programmu, lai analizētu steka operācijas tiešā assemblera līmenī. Šī metode identificē specifiskas steka operācijas AVR asemblera instrukciju kopā, ieskaitot push, pop, call, un steka rādītāju pielāgošanu ar sbiw/adiw instrukcijām. Tā palīdz identificēt arī funkciju izsaukumu grafu - kritisku informāciju steka dziļuma aprēķināšanai.

Pēc abu metožu izpildes, rīks izmanto abu metožu priekšrocības, lai iegūtu precīzāku novērtējumu. Šāda pieeja nodrošina, ka tiek ņemts vērā gan GCC konservatīvais steka izmantojuma novērtējums, gan tiešā asemblera koda analīze, izvēloties lielāko vērtību.

* + 1. **Rekursīvo funkciju analīze**

Rekursīvās funkcijas rada īpašu izaicinājumu steka analīzē, jo to steka izmantojums ir atkarīgs no rekursijas dziļuma. AVR steka analizators izmanto vairākas metodes, lai precīzi identificētu un analizētu rekursīvās funkcijas.

Divas metodes rekursīvo funkciju identificēšanai:

1. Pārbauda, vai funkcija tieši izsauc sevi.
2. Meklē raksturīgo rekursijas samazināšanas šablonu, kur parametrs tiek samazināts (piem., if (x > 0) { func(x - 1); }).

Tā kā rekursijas dziļums būtiski ietekmē steka izmantojumu, rīks analizē kodu, lai noteiktu maksimālo rekursijas dziļumu. Tas meklē tiešus funkcijas izsaukumus ar skaitliskām vērtībām. Identificē mainīgos, kas tiek padoti rekursīvām funkcijām. Analizē šo mainīgo sākotnējās vērtības. Nosaka iespējamo maksimālo rekursijas dziļumu.

Tas ir īpaši svarīgi, jo rekursijas dziļums tieši ietekmē steka izmantojumu - katra rekursīvā izsaukuma līmenī stekā tiek saglabāta funkcijas atgriešanās adrese un lokālie mainīgie.

Rīks ietver arī speciālu apstrādi funkcijai main(), kas bieži ietver bezgalīgus ciklus, kuri neietekmē steka dziļumu.

|  |
| --- |
| def is\_infinite\_loop\_function(self, func\_name):  """Nosaka, vai funkcija satur bezgalīgu ciklu."""  if func\_name != 'main':  return False  # Meklē funkcijas definīciju  func\_def\_pattern = re.compile(r'(?:\w+\s+)+' + re.escape(func\_name) + r'\s\*\([^)]\*\)\s\*{', re.DOTALL)  # [...]  # Meklē while(1) vai while(true) vai for(;;) modeli  infinite\_loop\_pattern = re.compile(r'while\s\*\(\s\*(1|true)\s\*\)|for\s\*\(\s\*;\s\*;\s\*\)')  if infinite\_loop\_pattern.search(func\_body):  return True  return False |

*4.1. koda fragments.* Bezgalīgā cikla noteikšanas funkcija

4.1. koda fragments identificē bezgalīgus ciklus, kas ir izplatīti iegulto sistēmu main() funkcijās. Šādi cikli netiek uzskatīti par "īstiem" rekursīviem izsaukumiem, jo tie nepalielina steka izmantojumu ar katru iterāciju.

* + 1. **Izsaukumu grafika rekonstrukcija no asemblera**

Precīza izsaukumu grafika rekonstrukcija ir būtiska maksimālā steka dziļuma aprēķināšanai. Rīks veic šo uzdevumu, analizējot gan asemblera kodu, gan pirmkodu. Izsaukumu grafika analīze tiek papildināta ar pirmkoda analīzi gadījumos, kad asemblera koda analīze sniedz nepilnīgu informāciju. Šis ir kritisks solis, jo daudzos gadījumos kompilators var optimizēt izsaukumus (piemēram, izmantojot funkciju ievietošanu), kas apgrūtina izsaukumu grafika rekonstrukciju tikai no asemblera koda.

Izsaukumu grafiks ir datu struktūra, kas attēlo, kā funkcijas programmas izpildes laikā izsauc citas funkcijas. Katrs grafika mezgls ir funkcija, bet šķautnes parāda izsaukuma attiecības starp tām. Steka analīzē šī informācija ir kritiski svarīga, jo:

* Steka pieaugums ir kumulatīvs - kad funkcija A izsauc funkciju B, funkcijai B nepieciešamā steka atmiņa tiek pievienota jau esošajam funkcijas A steka izmantojumam.
* Maksimālais steka izmantojums rodas specifiskā izpildes ceļā - tikai rekonstruējot pilnīgu izsaukumu grafiku, var identificēt tieši to programmas izpildes ceļu, kas izmanto visvairāk steka atmiņas.
* Rekursīvie izsaukumi ir īpaši nozīmīgi - grafika cikli norāda uz rekursīviem izsaukumiem, kur steka izmantojums var pieaugt proporcionāli rekursijas dziļumam.

AVR mikrokontrolieru arhitektūrā funkciju izsaukumi notiek, izmantojot specifiskas asemblera instrukcijas. Analizators atpazīst šīs instrukcijas, izmantojot regulāro izteiksmju meklēšanu disasemblētajā kodā. (4.2. koda fragments)

|  |
| --- |
| call\_pattern = re.compile(r'\s+(?:call|rcall|icall)\s+(?:0x)?([0-9a-f]+)?\s\*<?([^>\s]\*)')  for line in function\_code.split('\n'):  call\_match = call\_pattern.search(line)  if call\_match:  called\_function = call\_match.group(2)  if called\_function and called\_function not in call\_graph[function\_name] and called\_function in gcc\_stack\_usage:  call\_graph[function\_name].append(called\_function)  logger.debug(f"Found call from {function\_name} to {called\_function}") |

*4.2. koda fragments.* Funkciju izsaukumu atklāšana un grafu veidošana

4.2. koda fragments meklē trīs veidu izsaukuma instrukcijas AVR asemblerā:

* 1. **call** - absolūts izsaukums (izmanto absolūto adresi).
  2. **rcall** - relatīvs izsaukums (izmanto relatīvo nobīdi no pašreizējās adreses).
  3. **icall** - netiešs izsaukums (adresi norāda reģistrs, parasti izmanto funkciju norādēm).

Katra atpazītā izsaukuma instrukcija tiek analizēta, lai identificētu mērķa funkciju un atjauninātu izsaukumu grafiku.

Tiešo izsaukumu identificēšana (ar call un rcall instrukcijām) parasti ir vienkārša, jo mērķa funkcijas nosaukums vai adrese ir tieši norādīta instrukcijā. Sarežģītāks ir netiešo izsaukumu gadījums (ar icall instrukciju), kur mērķa funkcija nav tieši redzama. Šādos gadījumos analizators mēģina rekonstruēt izsaukuma mērķi, analizējot iepriekšējās instrukcijas, kas iestatījušas Z reģistra vērtību.

Netiešo izsaukumu identifikācija ir sarežģīta un ne vienmēr pilnībā iespējama tikai no asemblera koda, jo:

1. Z reģistra vērtība var tikt iestatīta dinamiski programmas izpildes laikā.
2. Funkciju norādes var tikt glabātas tabulās vai masīvos.
3. Var tikt izmantotas sarežģītas adresācijas shēmas.

Viena no lielākajām problēmām asemblera koda analīzē ir kompilatora optimizācijas, kas var būtiski mainīt sākotnējo koda struktūru:

1. **Funkciju ievietošana** - kompilators var aizstāt funkcijas izsaukumu ar pašas funkcijas kodu, pilnībā likvidējot izsaukuma instrukciju.
2. **Funkciju sapludināšana** - līdzīgas funkcijas var tikt apvienotas, lai samazinātu koda izmēru.
3. **Rekursijas transformācija** - rekursīvi izsaukumi var tikt pārveidoti par cikliem.
4. **Astes rekursijas optimizācija** - īpašs rekursijas veids, kur rekursīvais izsaukums ir funkcijas pēdējā darbība, var tikt optimizēts, lai neizmantotu papildu steka atmiņu.

Šīs optimizācijas var radīt situācijas, kur pirmkodā esošie funkciju izsaukumi asemblerā nav tieši redzami, radot nepilnīgu izsaukumu grafiku.

Lai risinātu asemblera koda analīzes ierobežojumus, analizators kombinē to ar pirmkoda (C valodas) analīzi:

|  |
| --- |
| known\_functions = list(gcc\_stack\_usage.keys())  if 'main' in call\_graph and not call\_graph['main']:  logger.warning("No calls detected from main(). Scanning source code for function calls.")  for func in known\_functions:  if func != 'main' and re.search(r'\b' + re.escape(func) + r'\s\*\(', self.source\_content):  call\_graph['main'].append(func)  logger.info(f"Found call from main to {func} in source code") |

*4.3. koda fragments.* Izsaukumu grafika papildināšana ar pirmkoda analīzi, kad asemblera analīze nesniedz rezultātus

4.3. koda fragments risina situāciju, kad asemblera koda analīze neatklāj visus 'main' funkcijas izsaukumus, parasti optimizāciju dēļ. Analizators tad pārslēdzas uz pirmkoda meklēšanu, identificējot funkciju izsaukumus, izmantojot regulārās izteiksmes.

Pirmkoda analīzē tiek izmantotas regulārās izteiksmes, lai atpazītu funkciju izsaukumus: **func\_call\_pattern = r'\b' + re.escape(func\_name) + r'\s\*\([^)]\*\)'**

Šī izteiksme atpazīst funkcijas nosaukumu kā atsevišķu vārdu (\b nodrošina vārda robežu), pēc kura seko neobligāta atstarpe (\s\*), tad atveroša iekava (\(), jebkuri simboli, izņemot aizverošo iekavu ([^)]\*) un nobeigumā aizverošā iekava (\)).

Lai pilnveidotu izsaukumu grafiku, analizators papildus meklē funkciju deklarācijas un definīcijas pirmkodā: **func\_def\_pattern = re.compile(r'(?:\w+\s+)+' + re.escape(func\_name) + r'\s\*\([^)]\*\)\s\*{', re.DOTALL).**

Šī izteiksme atpazīst funkcijas definīciju, kas sastāv no atribūtiem un atgriešanas tipa ((?:\w+\s+)+), funkcijas nosaukuma, parametru saraksta un funkcijas ķermeņa sākuma ar atverošo figūriekavu. Funkciju deklarāciju un definīciju identifikācija ļauj analizatoram izveidot pilnīgāku funkciju sarakstu, ko izmantot izsaukumu grafika rekonstrukcijā.

Lai atvieglotu analīzes rezultātu interpretāciju, analizators piedāvā izsaukumu grafika vizualizāciju DOT failā, kas var tikt pārveidots par grafisku attēlojumu, izmantojot rīkus kā Graphviz. Vizualizācija iezīmē rekursīvās funkcijas ar atšķirīgu krāsu un rekursīvos izsaukumus ar biezākām, sarkanām līnijām.

* + 1. **Maksimālā steka dziļuma aprēķināšana**

Maksimālā steka izmantojuma analīze apvieno visus iepriekšējos soļus vienotā algoritmā:

1. **Atsevišķi aprēķina rekursīvu funkciju steka izmantojumu**:

* Ņem vērā katra rekursijas līmeņa ietekmi uz steku.
* Ņem vērā atgriešanās adreses pārklājumu (2 baiti AVR arhitektūrā).
* Izmanto iepriekš noteiktos rekursijas dziļuma ierobežojumus.

1. **Izmanto dinamisko programmēšanu (memoizāciju)** steka izmantojuma aprēķināšanai:

* Saglabā jau aprēķinātos ceļus.
* Novērš atkārtotus aprēķinus sarežģītos izsaukumu grafos.
* Efektīvi apstrādā cikliskos grafus.

1. **Identificē maksimālo steka ceļu**:

* Katram funkciju izsaukumam atrod ceļu, kas izmanto visvairāk steka.
* Ņem vērā atgriešanās adreses pārklājumu.
* Apvieno vietējo steka izmantojumu ar izsaukuma ķēdes steka izmantojumu.

1. **Sāk analīzi no main() funkcijas**:

* Main() funkcija iegultajās sistēmās parasti ir izpildes steka pamatā, no kuras sākas visa programmas darbība.
* Nodrošina, ka tiek analizēti visi iespējamie izpildes ceļi.

Rīks pievieno drošības rezervi galīgajam steka izmantojuma novērtējumam. Šī rezerve ir būtiska AVR mikrokontrolieru programmēšanā, jo steka pārplūde var izraisīt grūti konstatējamas un prognozējamas kļūdas. 15% rezerve nodrošina papildu drošību pret neparedzētām situācijām un paaugstina programmas uzticamību.

Memoizācijas tehnika ļauj izvairīties no atkārtotas steka izmantojuma aprēķināšanas identiskiem izsaukuma ceļiem, kas ievērojami paātrina analīzi sarežģītiem izsaukumu grafikiem.

Rekursijas ierobežojumu dinamiska pielāgošanas pieeja ļauj modelēt rekursīvos izsaukumus izsaukumu grafikā, vienlaikus saglabājot iespēju ierobežot maksimālo rekursijas dziļumu, pamatojoties uz avota koda analīzi.

Rīks izmanto kombinētu pieeju, iegūstot izsaukumu grafiku gan no asemblera koda, gan no pirmkoda, lai nodrošinātu maksimālu precizitāti. (4.3. koda fragments)

Šī pieeja risina problēmu, kad funkciju izsaukumi tiek optimizēti kompilēšanas laikā, padarot izsaukumu grafika rekonstrukciju no asemblera koda vien nepilnīgu.

1. **Saīsinājumu un nosacīto apzīmējumu saraksts**

RISC - samazinātas instrukciju kopas dators

ALU - aritmētiski loģiskā ierīce

….

1. **Atsauces**

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

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

[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>

[6] <https://theswissbay.ch/pdf/Gentoomen%20Library/Misc/Art%20of%20Designing%20Embedded%20Systems~tqw~_darksiderg.pdf>

[7] <https://developerhelp.microchip.com/xwiki/bin/view/products/mcu-mpu/8-bit-avr/structure/memory/>