Sardsüsteemide programmi silumine

Erkki Moorits
Cybernetica AS, Navigatsioonisüsteemide osakond

Sardsüsteemides kasutatavad debugimise riistvara ja meetodid

- Simulaatorid
- JTAG
- Emulaatorid
- Kasutaja režiimis kernel
- Debug terminaalid
- Riistvaraliste signaalid
- Profileerimine ja koodi kattuvus

Simulaatorid vs. Emulaatorid

Järgnevalt on simulaatorite all mõeldud neid programme mis jäljendavad mittetäielikult protsessori või operatsioonisüsteemi tööd, näiteks ei võimalda reaalaja ülesandeid simuleerida või on muud moodi puudulikud. Seevastu emulaatorite all on üldiselt mõeldud riistvara mis jäljendab täpselt uuritavat süsteemi.

Simulaatorid

PIC Development Studio - E:\My Documents/PIC Development Studio projects/Example - 7 segment display.pds

Project | PIC Microcontroller | 7 Segment | Logic components

PIC Source Build Program CPU RAM MEM EE Watch Breaks Stack Watchdog

Program Memory

```
026:        clrf    PortB
027:        bsf     PortB,4
028: main   incf    PortB,r
029:        movf    PortB,w
02A:        sublw   0x1A
02B:        btfsc   Status,z
02C:        clrf    PortB
02D:        bcf     PortB,4
02E:        nop
02F:        nop
030:        bsf     PortB,4
031:        call    delay
032:        goto    main
033:        movlw   0x02
034:        movwf   TC0
035: _ls_0  movlw   0xff
036:        movwf   TC1
037: _ls_1  movlw   0xff
038:        movwf   TC2
039: _ls_2  decfsz  TC2,r
03A:        goto    _ls_2
03B:        decfsz  TC1,r
03C:        goto    _ls_1
03D:        decfsz  TC0,r
03E:        goto    _ls_0
03F:        return
```

Special Function Register Debug

		BIN	HEX	DEC	CHAR
	W	11111111	FF	255	ÿ
	Z	00000000	00	0	
	C	00000001	01	1	
0x00	INDF	00000000	00	0	
0x03	STATUS	00011001	19	25	□
0x04	FSR	00000000	00	0	□
0x05	PORTA	00000000	00	0	□
0x06	PORTB	00010001	11	17	□
0x08	EEDATA	00000000	00	0	
0x09	EEADR	00000000	00	0	
0x0B	INTCON	00000010	02	2	
0x85	TRISA	00000000	00	0	
0x86	TRISB	11100000	E0	224	

File Register Debug

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00	00	00	3A	19	00	00	11	00	00	00	00	02	00	00	00	00
10	01	47	E3	00	00	00	00	00	00	00	00	00	00	00	00	00
20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
30	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
40	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
50	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
60	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
70	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
80	00	FF	3A	19	00	00	E0	00	00	00	00	00	00	00	00	00
90	01	47	E3	00	00	00	00	00	00	00	00	00	00	00	00	00
A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

EEPROM Debug

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
10	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
30	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Stack Debug

0	031:	call	delay
1			
2			
3			

7 Seg

Logic 4511

A B C D E F G H
4511
1 2 3 4 E

PIC Microcontroller

PIC16F84-10/P
9851CAJ

Status: stopped | Project: E:\My Documents/PIC Development Studio projects/Example - 7 segment | Runtime: 4259841 (4259841) us | Speed: 55% of realtime (4 MHz)

CYBERNETICA

Simulaatorid

- Plussid
  - Kõige odavam viis riistvaralähedase programmi tööd kontrollida
  - Praktiliselt igale väiksemale kontrollerile olemas
  - Võimalik tarkvara ja riistvara koos simuleerida
- Miinused
  - Võib tekkida probleeme isegi simuleeritava riistvara enda eduka simuleerimisega
  - Võimaldab jälgida vaid neid programme, mis ei suhtle üldse või suhtlevad väga vähe ja aeglase välise riistvaraga
  - Aeglane

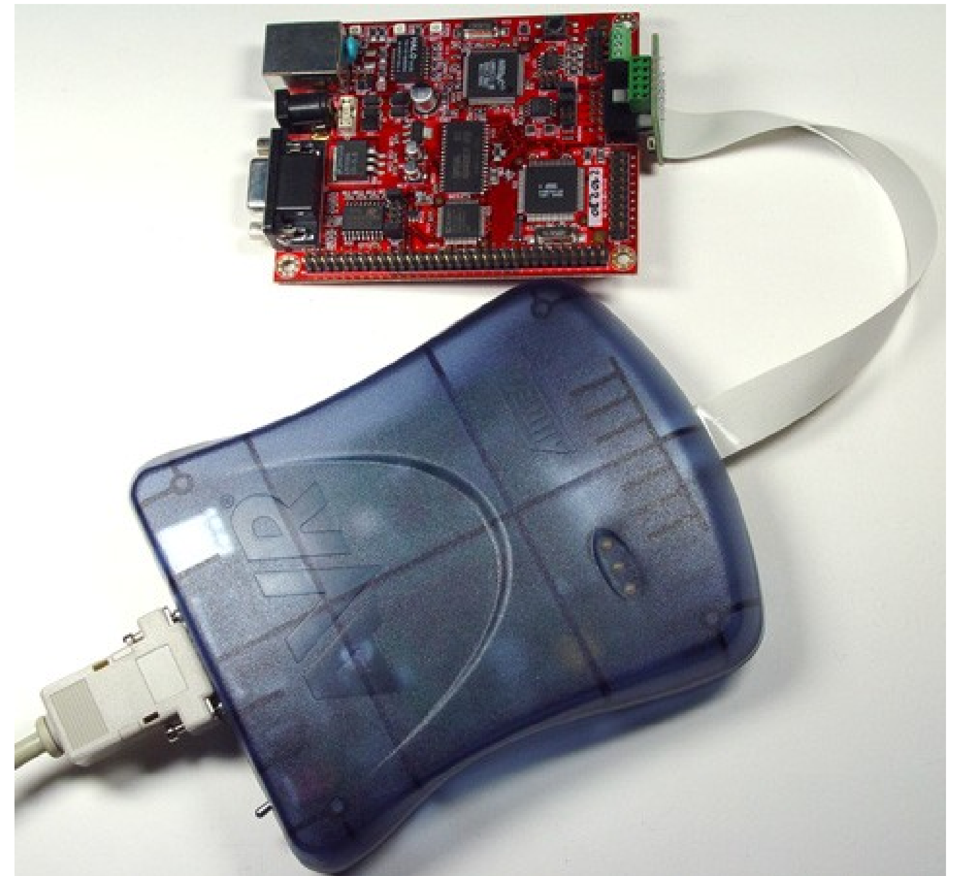
Tüüpiline simulatsiooni käik

- Kompileeritakse töötav programm, mis laetakse simulaatorisse, simulaator võib olla ka gdb osa. Selle sammu teevad ka kõik IDE'd (Integrated Development Environment)
- Kui simulatsiooni tulemus sõltub välistest signaalidest, siis tuleb eelnevalt kirjeldada kõiki väliseid signaale
- Vajadusel määrata simulatsiooni parameetrid ja katkestuskohad (breakpoint'id)
- Käivitatakse simulatsioon

Kus ja kuna kasutada simulaatorit

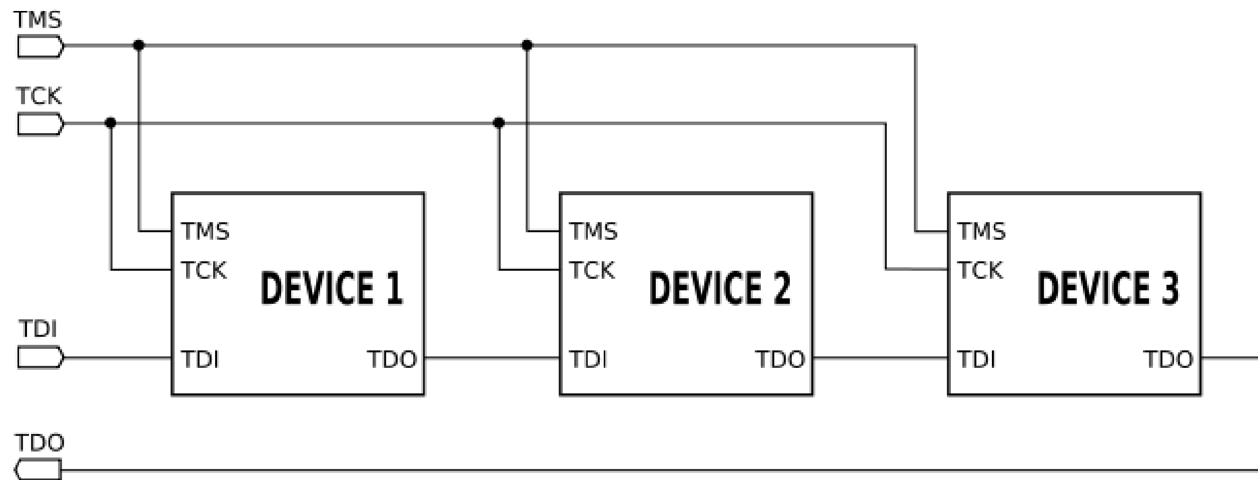
- Kui pole muid riistvaralisi silumise meetodeid
- Kui simuleerimine on tunduvalt ohutum või odavam kui reaalselt seadme tarkvara testimine
- Kui on vajadus tarkvara testida selliste sissendparameetrite juures mida muidu ei ole võimalik saada

JTAG (Joint Test Action Group, IEEE 1149.1)



JTAG

- Algselt loodud tootmisel seadmete kontrollimiseks, kuid sobib ka programmide silumiseks ja tarkvara laadimiseks
- On nõutud ainult 4 signaali (TDI, TDO, TCK, TMS), viies on valikuline (TRST), ehk tegemist on järjestik liidesega



CYBERNETICA

JTAG

- Miks JTAG on nii laialt levinud?
  - Üks odavamaid riistvaralisi meetodeid millega programmi tööd jälgida
  - Praktiliselt kõik protsessorid/mikrokontrollerid võimaldavad JTAG'iga silumist
- Probleemid JTAG'iga silumisel
  - Riistvara millel JTAG'i kasutatakse peab täpselt jälgima JTAG'i käske
  - Keerukas suuri programme kontrollida
  - Kõrge protsessori takti juures kasutamine raskendatud
  - Ei pruugi võimaldada protsessori sammude ajalugu talletada

Mida JTAG võimaldab

- Tarkvara laadimiseks kontrollerisse – seda suudavad kõik JTAG'i moodulid teha
- Kerneli käivitamisel toimuva jälgimiseks või mõne muu riistvara lähedase programmi silumiseks
  - Võimalik seada programmi hardware ja software breakpointe
- Tootmisel mikroskeemide ja nende ühenduste kontrollimist

Hardware vs. Software breakpoint

Software breakpoint

- *Software breakpoint* on protsessori peatamise instruktsioon mis on pannakse silumise ajal **ajutiselt** programm mällu. NB! Seadmetel millel on flash mälu lühendab selline tegevus flashi eluiga.

Hardware breakpoint

- Ülejäänud protsessorist sõltumatu elektroonika, mis jälgib protsessori tööd ja etteantud tingumusel peatab protsessori töö.

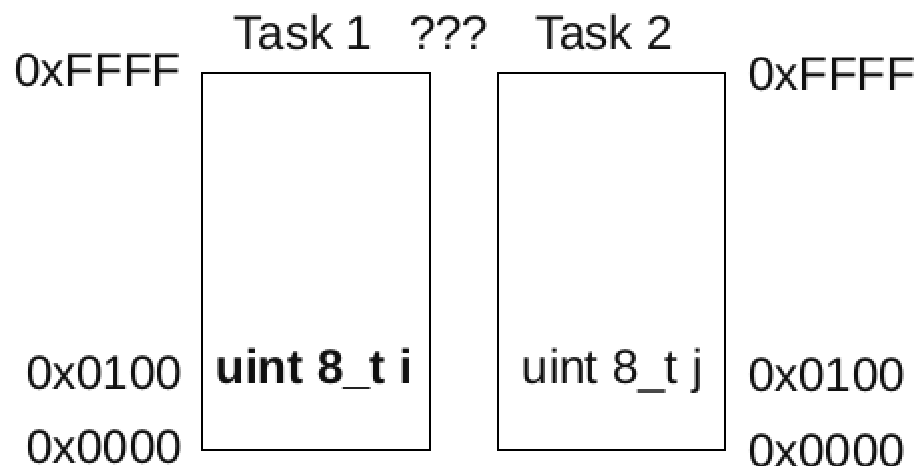
CYBERNETICA

JTAG'i käskude mittejälgimine

Mõningatel kontrolleritel ei pane JTAG kontrolleri asünkroon taimereid peale *breakpointi* saabumist koheselt pausile. See põhjustab omakorda asünkroon taimeril uue katkestuse ja ühtlasi muudab JTAG'iga silumise võimatuks.

JTAG ja MMU (Memory Management Unit)

- JTAG peab aru saama MMU omapäradest
  - MMU rakendamisel transleeritakse kõik füüsilised aadressid virtuaalseteks, mis kõik võivad alata samast aadressist – JTAG ei tohi “segadusse sattuda” nähes mitmes kohas sama aadressi (*run mode support*)



JTAG ja thread

- Süvasardsüsteemidel võib JTAG näidata vea asukohaks väga tihti kas scheduleri või siis mõnda perioodilist katkestust
  - Selline käitumine on iseloomulik sellistel juhtudel kui üks thread kirjutab teise threadi mälualasse mis omakorda põhjutab korrasolevas threadis vea. MMU olemasolu üldiselt välistab sellise võimaluse.

CYBERNETICA

JTAG'i kasutamine

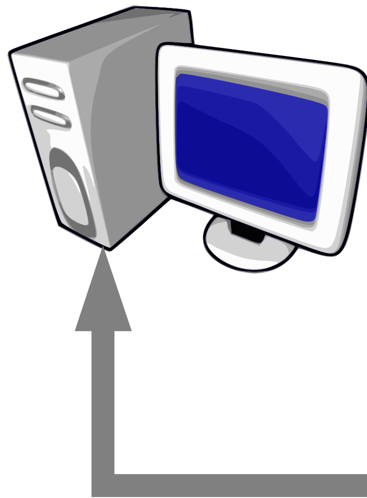
- JTAG on kasutatav ainult koos vastava debugeriga, enamusjaolt on selleks *gdb* (GNU Debugger)
- JTAG'i kasutamine on väga sarnane simulatsiooniga, *gdb* puhul on ainukeseks erinevuseks on debugeri teine back-end
  - Näiteks AVR mikrokontrollerite puhul tuleb ühendada gdb parasjagu käiva AVaRICE programmiga. Väga hea samm sammult näide on aadressil: http://winavr.sourceforge.net/AVR-GDB_and_AVaRICE_Guide.pdf

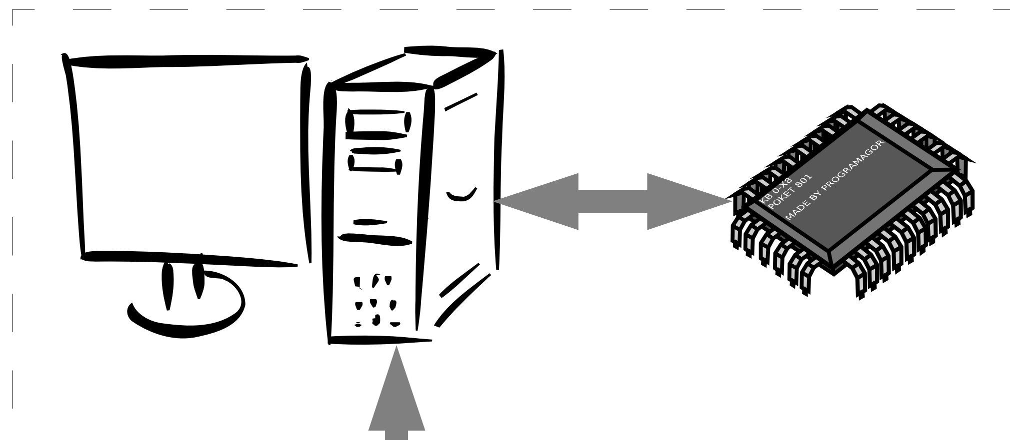
CYBERNETICA

Tüüpiline JTAG'i ühendus

Arvuti koos gdb'ga

Silutav süsteem (koos JTAG'iga)





Debugeri ja silutava süsteemi vaheline link

- Silutavat süsteemi ja silujat ühendav link võib olla järjestikliides, IEEE 1394 (FireWire), ethernet, jne
- Silutav süsteem võib olla ka simulaator või emulaator

JTAG'i kasutuskohad

- Arenduses programmi töö debugimiseks
  - Üldiselt võimalik kuni scheduleri käivitamiseni
- Tootmises seadme elektroonika korrasoleku kontrolliks

Emulaatorid

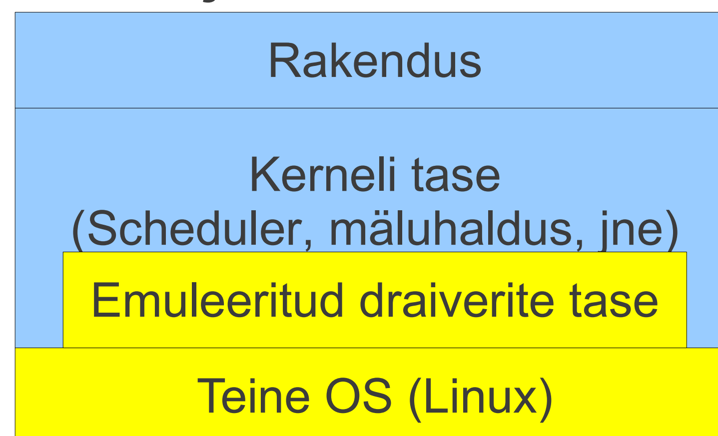
- Riistvaraliste emulaatorite eelised
  - Käituvad täpselt nagu emuleeritav riistvara
  - Võimalik üpriski pikka protsessori sammude ajalugu salvestada
- Riistvaraliste emulaatorite puudused
  - Vähemalt suurusjärgu võrra kallim kui JTAG
  - Uutele kiiretele protsessoritele/mikrokontrolleritele ei pruugi neid olla, ega ka tulla
  - Ei pruugi samasuguste elektriliste parameetritega olla nagu on emuleeritav seade ise
  - Rakendusprogramme ikkagi keerukas debugida

Emulaatorite kasutuskohad

- Kernelite ja muu riistvaralähedase koodi debugimiseks nendes kohtades kus jääb JTAG'i võimalustest puudu
  - Võimaldab salvestada sammude ajalugu

Kasutaja režiimis kernel

Kasutaja režiimi kernel on operatsioonisüteemi teekide ja draiverite kogu, mis on kompileeritud nii, et ei suhtle otse riistvaraga vaid suhtleb läbi teise OS'i. Linuxi puhul kutsutakse seda ka User-Mode Linux (UML)



Kasutaja režiimis kernel

- Kasutaja režiimis kerneli/programmi kasutuskohad
  - Rakendusprogrammide kontroll
  - Kerneli mitte riistvaraga suhtlevate osade kontroll
  - Linuxi puhul ühe riistvaralise masina peale mitme turvalise süsteemi loomine
- Puudused
  - Ei võimalda riistvaralähedasi programme emuleerida
  - Väga väheseid kerneleid annab kasutaja režiimis kompileerida
  - Töökiirus sõltub peremees operatsioonisüsteemist

Kus kasutada kasutaja režiimis kernelit

- Programmi kontseptsiooni või idee kiireks kontrolliks ja realiseeritavust kasutatava operatsioonisüsteemiga
- Rakendusprogrammide debugimiseks
- Kerneli poolt pakutavate funktsioonide kontrolliks

CYBERNETICA

Debug terminaal

Debug terminaaliks kutsutaks ühte väljundporti, mis on reeglina järjestikport, kuhu kontroller saadab oma vea või olekuteateid. Unix'i laadsetel operatsioonisüsteemidel (Linux, *BSD) on samade ülesannetega väljundiks süsteemi konsool (*system console*). Debug terminaali ülesandeid võib mõnikord täita ka teenindusprogramm.

CYBERNETICA

Debug terminaalid

- Miks kasutada debug terminale
  - Hinna ja efektiivsuse suhtelt kõige parem
  - Võimaldab praktiliselt terve kerneli tööd jälgida

- Puudused
  - Mõningatel juhtudel võib olulisel määral mõjutada programmi tööd
  - Ajakriitilistes või piiratud mäluga kohtades ei ole võimalik kasutada
  - Riistvaral ei pruugi olla ühtegi vaba väljundporti (UART, USB, jne.)

Debug terminali kasutamine – printf() debugging

```
#define DEBUG_OUT stdout
#endif

#if (HAS_TRACE == 1)
#define TRACE fprintf (DEBUG_OUT, "%s @ %u\n", __FILE__, __LINE__)
#else
#define TRACE {}
#endif

int main (void)
{
    puts ("Hello world!");
    TRACE;
    puts ("Hello world! Again");
    return 0;
};

/*---- väljund ----*/
Hello world!
example-debugterm.c @ 16
Hello world! Again
```


CYBERNETICA

Linuxi kerneli printk

- Spetsiaalne käsk Linuxi kerneli teadete väljastamiseks
  - Sama formaadiga kui tavaline printf
  - Võimaldab lisada erinevaid logimise tasemeid

```
printk(KERN_WARNING "This is a warning!\n");

printk(KERN_DEBUG "This is a debug notice!\n");

printk("I did not specify a loglevel!\n");
```


CYBERNETICA

Printf silumise miinuskohad

- Enamusjaolt on ad hoc silumise meetodiga – vea avastamiseks lisatakse käske ja peale vea leidmist eemaldatakse
- Väljund on tavaliselt puhvris ja programmi kokkujooksmisel ei jää midagi alles
  - Mõndadel kernelitel on spetsiaalne UART debug draiver millel puudub puhber ja töötab pollimisega, aga seda tüüpi draiver mõjutab oluliselt programmi tööd
- Mõjutab kompileerimisel registrite ja mälu kasutust
- Aeglustab programmi täitmist

Mõned märkused debug terminalide kohta

- Debug terminali ei saa kasutada või on kasutamine raskendatud järgnevates kohtades:
  - Kerneli käivitamise alguses
  - Enamustes bootloaderites
  - Katkestustes

- Mikrokontrolleritel, seda eriti Harvardi arhitektuuriga seadmetel võib hooletu printf käskude lisamine võtta märkmisväärse osa mälust

Kus kasutada debug terminale

- Lõplikus tootes süsteemi seadistamiseks ja töö jälgimiseks
- Arenduse käigus debugimiseks

Assertions

- Assertions on koodi kontrolli lõigud mis peavad olema tõese väärtusega teatud koodi kohas
- Mittetõese väärtuse puhul katkestatakse programmi töö
- Sardsüteemidel raskesti kasutatav
  - Tavaliselt võtab liiga palju resurssi

```
#include <assert.h>

void foo (uint8_t x)
{
    assert(e > 3)
   /* kood */
}
```

Kompilaatori poolne optimeerimine ja programmi silumine

- Praktiliselt kõik kompilaatori poolsed optimeerimised teevad programmi raskemini silutavaks
  - Silumise käigus tuleb väga paljudel juhtudel vaadata programmi täitmist assembleris, optimeerimine aga paigutab koodi üpriski suurel määral ümber
- Üks võimalus optimeeritud programmist spetsiifilise koodilõigu leidmiseks on nop instruktsioonide lisamine
  - Ei toimi sellisel riistvaral kus kasutatakse suht palju nop instruktsioone (PC)

CYBERNETICA

nop'i kasutamine

```
#include <stdio.h>
int main (void)
{
    asm volatile ("nop");
    printf ("hello world\n");
    asm volatile ("nop");
    return 0;
}
```

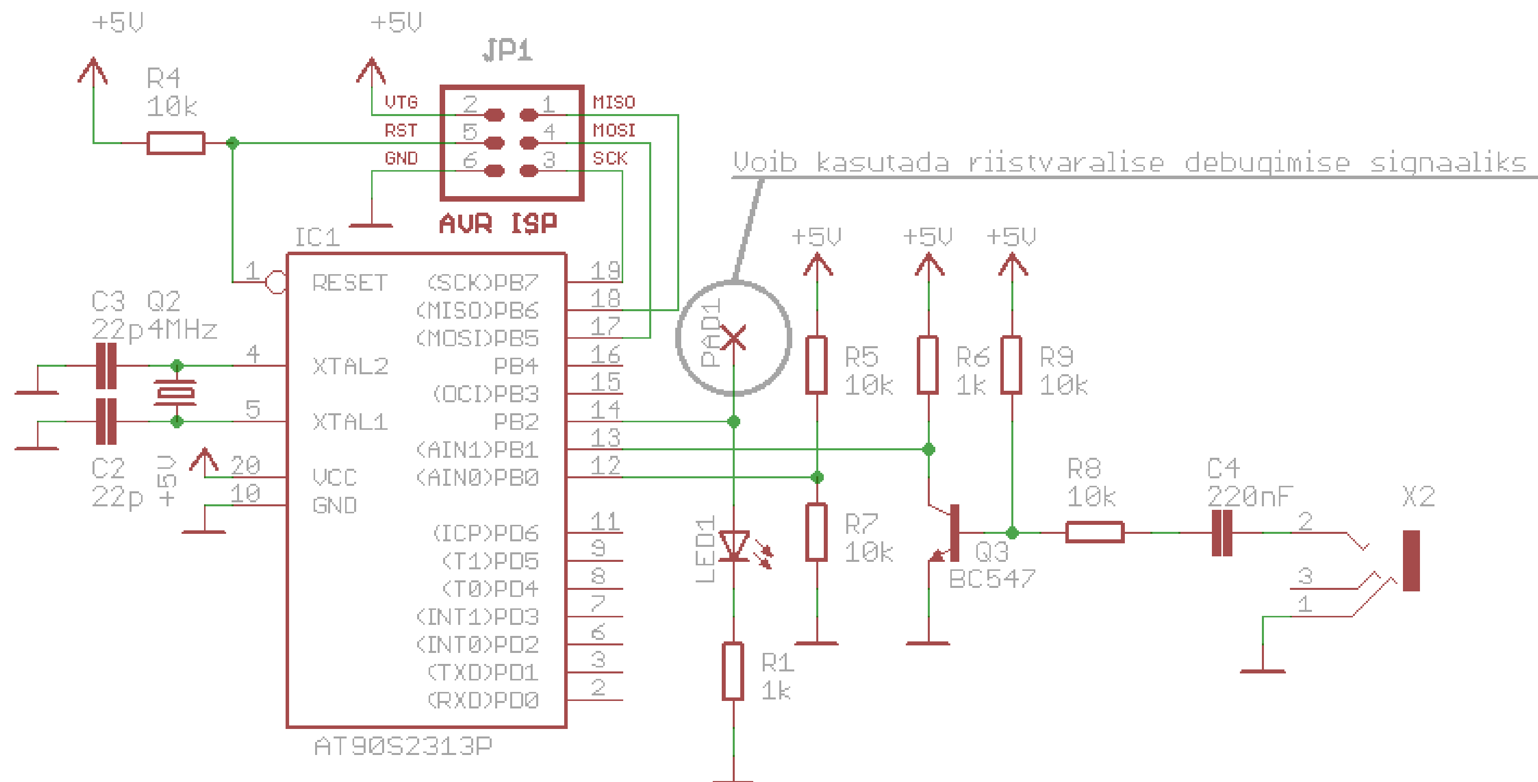
```
00000028 <main>:
  28:   df 93           push    r29
  2a:   cf 93           push    r28
  2c:   cd b7           in  r28, 0x3d   ; 61
  2e:   de b7           in  r29, 0x3e   ; 62
  30:   00 00           nop
  32:   80 e6           ldi r24, 0x60   ; 96
  34:   90 e0           ldi r25, 0x00   ; 0
  36:   06 d0           rcall   .+12        ; 0x44 <puts>
  38:   00 00           nop
  3a:   80 e0           ldi r24, 0x00   ; 0
  3c:   90 e0           ldi r25, 0x00   ; 0
  3e:   cf 91           pop r28
  40:   df 91           pop r29
  42:   08 95           ret
```

Unit testid

- Sardsüsteemidele on tehtud mõned üksikud unit testid (embunit), kuid nende kastamine on üpriski tülikas tegevus
  - Käib ainult sellel riistvaral millele kirjutatakse parjasjagu programmi
  - Nõuab iga uue testiga ka uue programmi sisselaadimist

CYBERNETICA

Riistvaralised signaalid



Riistvaralised signaalid

- Riistvaraliste signaalide eelised
  - Paljudel juhtudel ei maksa ühe signaali väljatoomine eriti palju
  - Enamustel süsteemidel on olemas mingi oleku indikaator või mõni vaba väljund mida saab vastavalt programmi olekule lülitada
  - Vastava aparatuuri olemasolul annab väga kiireid programmilõike jälgida, isegi schedulere
- Puudused
  - Riistvaraliste signaalidega annab kontrollida ainult lihtsate programmilõikude tööd
  - Programmi debugimine on väga aeganõudev
  - Keerukatel süsteemidel raske rakendada

Kus kasutada riisvaraliste signaalide järgi debugimist

- Väikestes või hobi projektides programmi töö kontrolliks
- Lõplikus tootes süsteemi üldise oleku näitamiseks, kas töötab või ei tööta
- Tootearenduses mõne väga ajakriitilise programmilõigu jälgimisel
- Piiratud võimalustega mikrokontrolleri programmi debugimisel (näiteks PIC10F200, mis on SOT23-6 korpuses)

Profileerimine

Profiling allows you to learn where your program spent its time and which functions called which other functions while it was executing. This information can show you which pieces of your program are slower than you expected, and might be candidates for rewriting to make your program execute faster. It can also tell you which functions are being called more or less often than you expected. This may help you spot bugs that had otherwise been unnoticed.

(http://sourceware.org/binutils/docs-2.20/gprof/index.html)

Profileerimine sardsüsteemides

- 8 ja 16 bitistel mikrokontrolleritel ning isegi väiksematel ARM'idel ei ole võimalik kasutada enamlevinud profileerimise rakendusi, näiteks *gprof*'i

- Alternatiivsed variandid:
  - Profileerimiseks on võimalik kasutada simulaatoreid, mis programmi täites loendavad etteantud sündmusi
  - Võimalik kirjutada kerneli koodi sisse vastavaid profileerimise funktsioone ja pärast väljastada tulemus debug porti või mujale väljundisse

Koodi kattuvuse kontroll

- Teadaolevalt ei ole veel tehtud süvasardsüsteemidele ühtegi tõsiseltvõetavat koodikattuvuse kontrolli süsteemi
- Koodikattuvuse kontrolli on võimalik sisuliselt kahte moodi teha:
  - Vastavate trace käskudega, mis edastatakse kas üle JTAG'i või siis üle seriaal liidese
  - Simulaatori abil mis väljastab käimasoleva simulatsiooni käigu koodikattuvuse arvestuse programmi

CYBERNETICA

Debugimis meetodite võrdlus

Meetod / Kasutuskoht	Kerneli debugimine	Rakenduse debugimine	Mõju seadme tööle
Simulaator	Enamusjaolt võimalik	Aega nõudev	Puudub
JTAG	Enamusjaolt võimalik	Raskendatud või ebaefektiivne	Vähestel juhtudel võib mõjutada
Emulaator	Võimalik	Raskendatud või ebaefektiivne	Praktiliselt olematu
Kasutaja režiimis kernel	Osaliselt võimalik	Võimalik	Puudub
Debug terminaal	Enamusjaolt võimalik	Võimalik	Oleneb kasutuskohast, mõnikord võib olulisel määral mõjutada
Riistvaralised signaalid	Võimalik, kuid võib olla ebaefektiivne	Praktiliselt võimatu või ebaefektiivne	Minimaalne
Profileerimine	Võimalik, kuid võib põhjustada arusaamatuid viiteid	Võimalik	Oleneb kasutuskohast, mõnikord võib olulisel määral mõjutada

Kokkuvõte

- Debugimis meetodi valik sõltub väga olulisel määral süsteemist ja rakendusese mida debugida soovitakse
- Praktiliselt kõik debugimise meetodeid võivad mingil määral mõjutada süsteemi toimimist
- Arenduses sobivad debugimiseks kõige paremini JTAG ja debug terminaal
- Tootmises sõltub debugimise meetod seadme eripärast, võivad sobida nii lihtne riistvaraline signaal, JTAG ja debug terminaal
- Väljastatud tootes sobivad kõige paremini seadme kontrolliks lihtne riistvaraline signaal ja debug terminaal

Küsimusi?

CYBERNETICA