nedeljainformatike v8.0

Kako pronalazimo greške u programima?

Lazar Premović

Matematička gimnazija

15. 05. 2023.

O čemu NEĆEMO pričati

- Formalna verifikacija
- Statička analiza koda

Agenda

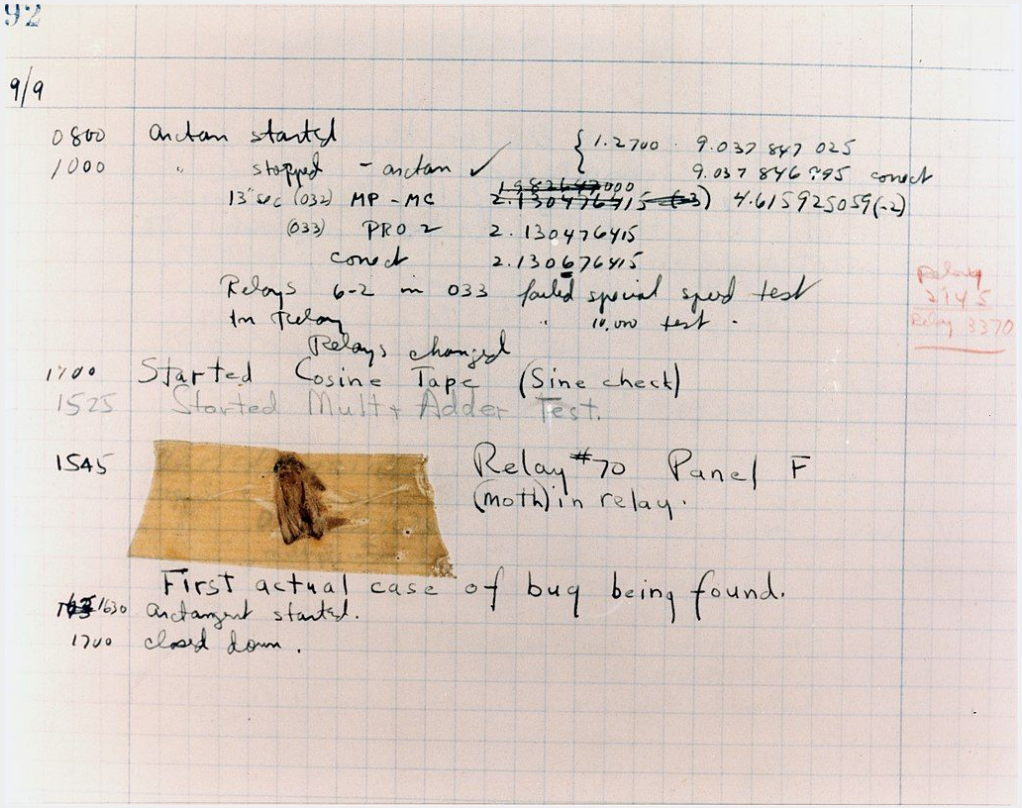
1. Šta je debager i njihova istorija
2. Kako rade debageri
3. Alternative i saveti

Debager

- Program je niz instrukcija
- Kada se program izvršava, pored instrukcija on poseduje i neko stanje
- Pri izvršavanju programa, procesor:
  - Određuje koju instrukciju treba da izvrši na osnovu trenutnog stanja
  - Dohvata tu instrukciju
  - Izvršava tu instrukciju i time dovodi program u novo stanje
- Stanje programa se nalazi u radnoj memoriji i u registrima procesora
- Posao debagera je da nam omogući da vidimo to stanje dok se program izvršava
- Debager takođe treba da nam omogući da nekako kontrolišemo izvršavanje programa, u suprotnom ne bi baš stigli da vidimo šta se dešava sa stanjem

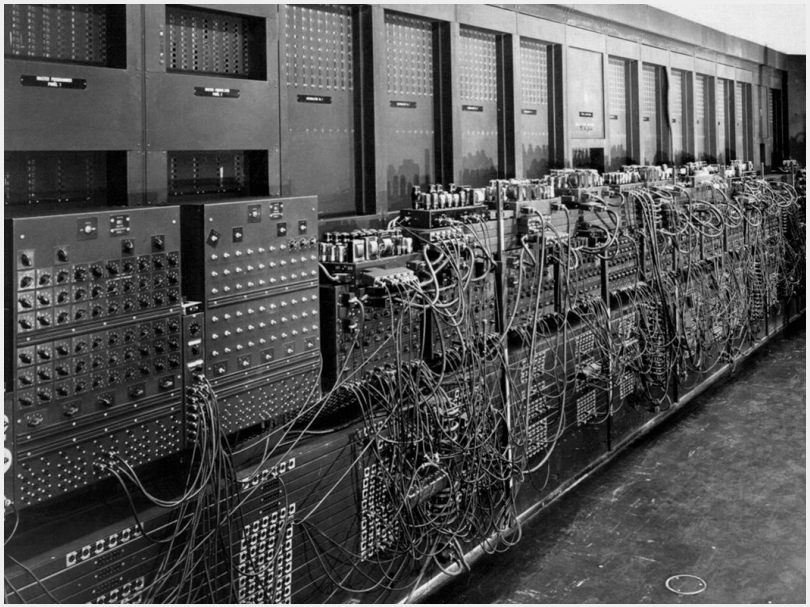
Prvi “bag”

- Izraz “bag” se koristio u inženjerskom žargonu još od 1870ih
- Prva upotreba izraza “bag” u kontekstu računarskog sistema se pojavljuje 1947 kada je razlog otkaza elektromehaničkog računara “Harvard Mark II” bio moljac koji se zaglavio u jednom od releja

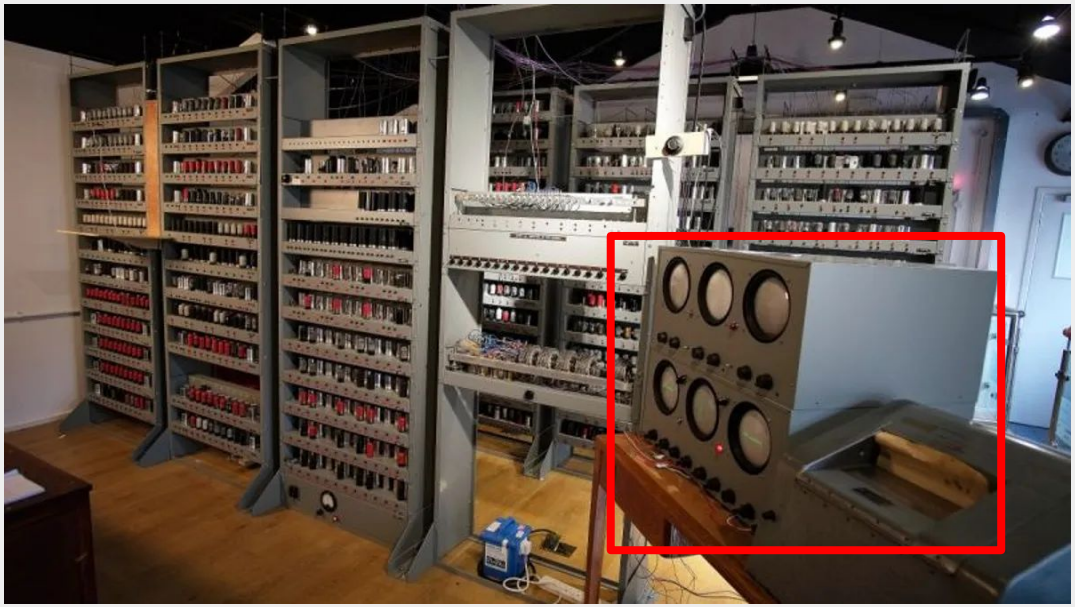


Debagovanje prvih računara

- Kako su debagovani programi na prvim računarima?
- JAKO TEŠKO
- Šta je minimalno potrebno za debagovanje?
- Mogućnost da se program izvršava instrukciju po instrukciju
- Lampice (ili osciloskopi) koji prikazuju stanja bitova u memoriji



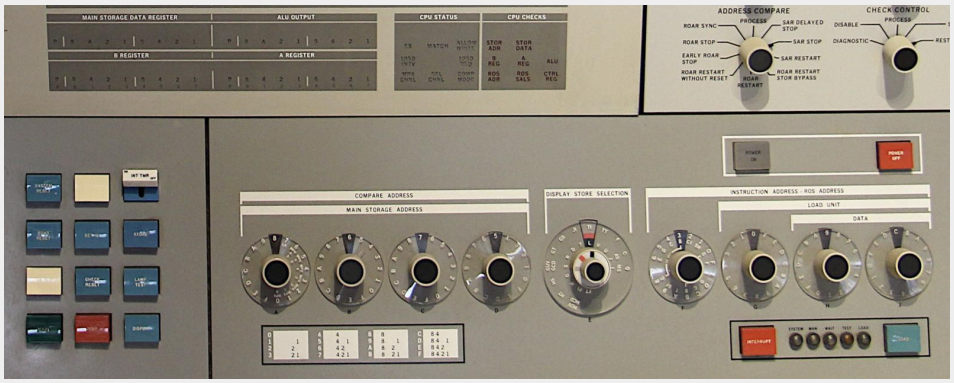
ENIAC 1946

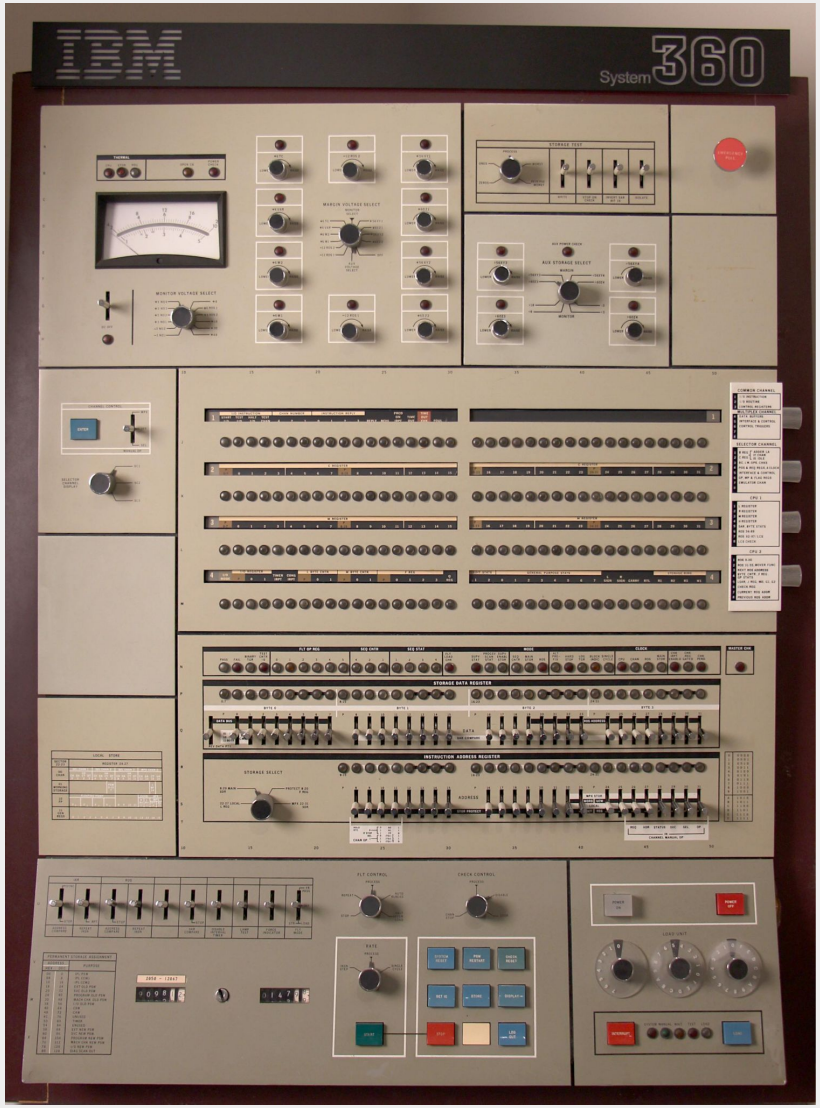


EDSAC 1949

Debagovanje prvih računara

- Ubrzo nakon lampica je dodata i mogućnost da se program izvršava punom brzinom do neke instrukcije i da se pri nailasku na tu instrukciju zaustavi, što je preteča modernih tačaka prekida (*breakpoints*)
- Za razliku od modernih tačaka prekida, ova mogućnost je realizovana u hardveru i ograničena je na mali broj tačaka prekida (često samo jedna)
- Ove metode debagovanja su ostale zastupljene dosta dugo (barem kao poslednja opcija)



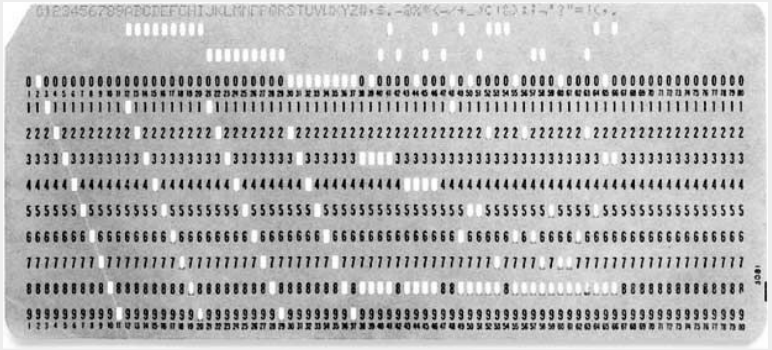


IBM SYSTEM/360 MODEL 30/50 1964

Kartice i terminali

- Računari su ubrzo počeli da koriste bušene kartice za učitavanje programa kao i linijske štampače za ispis rezultata, te je postalo moguće i relativno efikasno ispisivati stanje programa ili podatke od interesa iz samog programa na mestima gde je to poželjno (preteča *print debugging*-a)

- Sledeća bitna inovacija su bili terminali i rasprostranjenost time-sharing sistema, ovo je omogućilo nastanak interaktivnih debagera koji se koriste iz komandne linije (Dynamic Debugging Technique - 1961)



Simbolički debageri

- Trenutno sa interaktivnim debagerima imamo mogućnost da pročitamo ili upišemo bilo koju vrednost u memoriju i izvršavamo naš program instrukciju po instrukciju kroz terminal (više nemamo pristup hardverskim tačkama prekida jer terminali ne moraju biti pored samog računara)

- Sada se već uveliko koriste viši programski jezici ali pri debagovanju programeri moraju da znaju da se njihova promenljiva A nalazi na adresi 0x5372 ili da instrukcija na adresi 0x0092 odgovara 73. liniji izvornog koda

- Simbolički debageri rešavaju ovaj problem generisanjem dodatnih podataka pri prevođenju izvornog koda

- Ovi podaci mapiraju promenljve na njihove adrese kao i linije izvornog koda na adrese instrukcija

- Debager kasnije koristi te podatke kako bi omogućio programeru da koristi imena promenljivih iz izvornog koda i postavlja tačke prekida koristeći brojeve linija izvornog koda

Softverske tačke prekida

- Kako su tačke prekida veoma korisne i izvršavati instrukciju po instrukciju nije prihvatljivo za veće programe, interaktivni debageri su morali da implementiraju tačke prekida na neki način

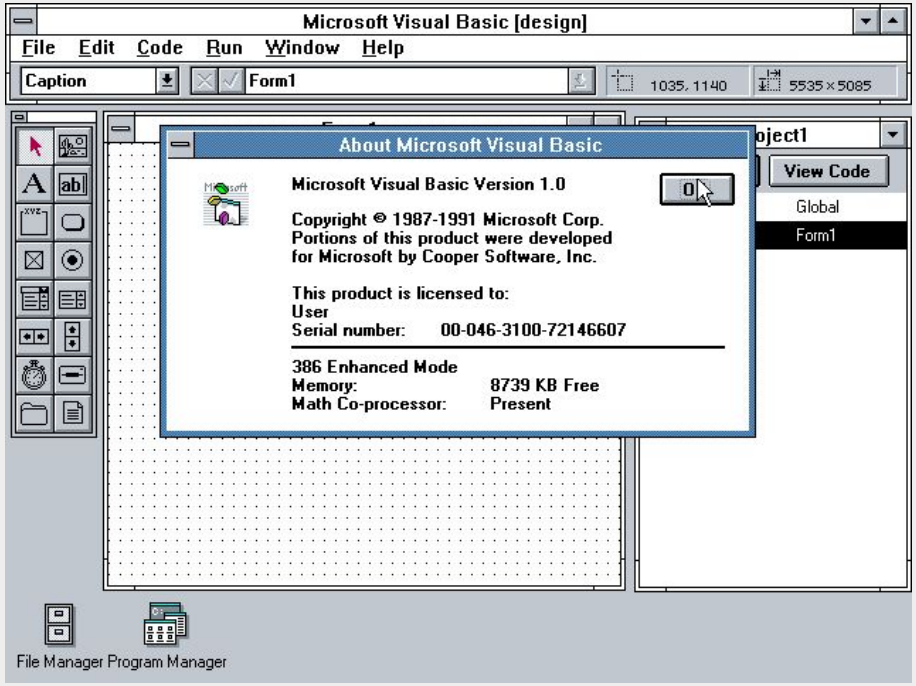
- Softverske tačke prekida su alternativa hardverskim tačkama prekida i bazirane su na konceptu prekida (nešto više o tome kako rade će biti rečeno kasnije)

- Pored toga što ne zavise od fizičkog pristupa hardveru, u jednom trenutku može postojati neograničen broj softverskih tački prekida

- Ovo takođe omogućava debageru da pri nailasku na tačku prekida proveri neki uslov, i ukoliko on nije ispunjen automatski nastavi izvršavanje programa, efektivno kreirajući uslovne tačke prekida (*conditional breakpoints*)

Integrisano razvojno okruženje

- Interaktivni debageri sada imaju većinu funkcionalnosti koje očekujemo od modernog debagera
- Jedino što je preostalo je integrisati funkcionalnosti debagera sa ostalim funkcionalnostima koje programeri koriste pri razvoju softvera u jedan softverski paket
- Jedna od prvih integrisanih razvojnih okruženja su bila
  - Microsoft Visual Basic (1991)
  - Borland TurboPascal (1993)



Agenda

1. Šta je debager i njihova istorija
2. Kako rade debageri
3. Alternative i saveti

Moderni debageri

- Od modernih debagera se očekuje da mogu da:
  - Manipulišu memorijom
  - Upravljaju izvršavanjem instrukcija
  - Omoguće upotrebu običnih i uslovnih tačaka prekida
  - Omoguće upotrebu simbola umesto adresa
  - ...

Manipulacija memorijom

- Načelno prosta stvar, ukoliko pristupamo svojoj memoriji
- Izvršavamo ciljni program kao deo debagera
- Nije preterano korisno, ako ciljni program pukne i debager se gasi
- Rešenje: debager i ciljni program su odvojeni procesi
- Ovo uslovljava postojanje operativnog sistema (samim tim i zaštite memorije)
- Rešenje: operativni sistem pruža mehanizam pristupa memoriji drugog procesa, specifično radi debagovanja
- Linux: ptrace Windows:Win32 Debug API

Kontrola toka

- Kontrola toka od strane debagera se bazira na konceptu softverskih prekida
- Softverski prekidi liče na skokove ili pozive funkcija, sa razlikom da destinaciju skoka specifikuje neko drugi (uglavnom operativni sistem)
- Kako operativni sistem upravlja prekidima, opet je potrebno da koristimo mehanizam operativnog sistema kako bi upravljali kontrolom toka

Izvršavanje instrukciju po instrukciju

- Na visokom nivou:
  - Nakon što pokrenemo ciljni program pod kontrolom debagera, ciljni program je blokiran i debager može da izda komandu operativnom sistemu da izvrši jednu instrukciju ciljnog programa, nakon toga ciljni program se opet blokira i debager se obaveštava o tome

- Na niskom nivou:
  - Skoro svi moderni procesori podržavaju *trap* fleg, kada je ovaj fleg aktivan procesor će nakon svake izvršene instrukcije generisati prekid i izvršiti odgovarajuću prekidnu rutinu, unutar te prekidne rutine operativni sistem blokira ciljni proces i prebacuje kontrolu debageru

Tačke prekida

- Šta ako želimo da do prekida (tj. prebacivanja kontrole dođe samo nakon određene instrukcije?
- Pored trap prekida postoje i instrukcije koje izazivaju određeni prekid
- Na x86 to je instrukcija INT (opcode 0xCD xx) ili INT3 (opcode 0xCC)
- Za debagovanje se skoro isključivo koristi INT3
- Zašto?
- Biblija tako kaže...

  The INT 3 instruction generates a special one byte opcode (CC) that is intended for calling the debug exception handler. (This one byte form is valuable because it can be used to replace the first byte of any instruction with a breakpoint, including other one byte instructions, without overwriting other code).

  Intel's Architecture software developer's manual, volume 2A

Tačke prekida

- Znači želimo da ciljni program umesto instrukcije na koju smo stavili tačku prekida izvrši INT3
- Potrebno zameniti prvi bajt ciljne instrukcije sa 0xCC
- Kada ciljni dođe do te instrukcije:
  - umesto nje izvrši INT3
  - operativni sistem predaje kontrolu debageru
  - debager vraća orginalnu vrednost bajta na koji je upisao 0xCC
  - debager vraća PC jedan bajt unazad
  - debager daje kontrolu korisniku
- Pri nastavku izvršavanja (ukoliko tačka prekida i dalje treba da postoji):
  - debager izvršava jednu instrukciju ciljnog programa
  - debager opet zamenjuje prvi bajt prethodne instrukcije sa 0xCC

Uslovne tačke prekida

- Kod uslovnih tačaka prekida, pre predaje kontrole korisniku debager proverava uslov koji odgovara toj tački prekida i ukoliko on nije ispunjen automatski nastavlja izvršavanje bez uklanjanja te tačke prekida
- Kako bi ovo podržala, većina debagera može da evaluira logičke izraze
- Po pravilu, ovi izrazi zavise od stanja programa ali njihova evaluacija ne bi trebala da utiče na samo stanje programa
- Neki debageri imaju i dodatne opcije za uslovne tačke prekida
  Npr. zaustavljanje programa na svaki N-ti prolaz kroz tačku prekida

Simboličko debagovanje

- Kao što smo već spomenuli, kako bi debageri mogli da rade sa imenima promenljivih iz izvornog koda umesto sa memorijskim adresama, potrebno je da prilikom prevođenja generišemo dodatne podatke koji omogućavaju to mapiranje
-
- Ti dodatni podaci takođe sadrže i mapiranje između linija izvornog koda i adresa instrukcija
-
- Postoji nekoliko različitih formata koji mogu da čuvaju te dodatne podatke, najčešće se koriste DWARF (Linux) i PDB (Windows)

Program DataBase

- Program DataBase (PDB) je format koji Microsoft prevodioci koriste za smeštanje debug podataka
- Sam format .pdb fajlova nije javno dostupan ali Microsoft obezbeđuje Debug Interface Access (DIA) API koji se koristi za pristup debug informacijama unutar .pdb fajlova
- *msdia80.dll* *msdia100.dll* *msdia140.dll*

DWARF

- DWARF je format za čuvanje debug podataka originalno dizajniran da se koristi uz ELF (format za objektne fajlove) iako je suštinski nezavistan od formata objektnog fajla
- Trenutno je aktuelna verzija 5 iz 2017.
- DWARF čuva sve podatke u strukturi pod nazivom Debugging Information Entry (DIE), jedan DIE predstavlja jednu promenljivu, tip, funkciju...
- Svaki DIE ima svoj tip (*DW_TAG_variable*, *DW_TAG_pointer_type*, *DW_TAG_subprogram*, ...), attribute (par ključ vrednost) kao i ugnežđene DIE-ove tako da formiraju stablo
- Atributi takođe mogu da referenciraju druge DIE-ove koji se nalaze bilo gde u stablu

DWARF - Tabele mapiranja linija i okvira funkcija

- Tabele koje mapiraju linije izvornog koda na adrese instrukcija zauzimaju dosta prostora te se najčešće kompresuju
- U DWARF-u te tabele su predstavljene kao niz instrukcija za posebnu mašinu stanja
- Sličan algoritam sa mašinom stanja se takođe koristi i za tabelu okvira funkcija koja govori debageru kako funkcija izgleda na steku u svakom trenutku njenog izvršavanja (ovo omogućava debageru da prikaze istoriju poziva funkcija, *call-stack*)

DWARF - Primer

```c
#include <stdio.h>


void do_stuff(int my_arg)
{
   int my_local = my_arg + 2;
   int i;

   for (i = 0; i < my_local; ++i)
      printf("i = %d\n", i);
}


int main()
{
   do_stuff(2);
   return 0;
 }
```

How debuggers work - Eli Bendersky (eli.thegreenplace.net)

```
.debug_info
<1><71>: Abbrev Number: 5 (DW_TAG_subprogram)
   <72>   DW_AT_external    : 1
   <73>   DW_AT_name        : (...): do_stuff
   <77>   DW_AT_decl_file   : 1
   <78>   DW_AT_decl_line   : 4
   <79>   DW_AT_prototyped  : 1
   <7a>   DW_AT_low_pc      : 0x8048604
   <7e>   DW_AT_high_pc     : 0x804863e
   <82>   DW_AT_frame_base  : 0x0 (loc. list)
   <86>   DW_AT_sibling     : <0xb3>

<1><b3>: Abbrev Number: 9 (DW_TAG_subprogram)
   <b4>   DW_AT_external    : 1
   <b5>   DW_AT_name        : (...): main
   <b9>   DW_AT_decl_file   : 1
   <ba>   DW_AT_decl_line   : 14
   <bb>   DW_AT_type        : <0x4b>
   <bf>   DW_AT_low_pc      : 0x804863e
   <c3>   DW_AT_high_pc     : 0x804865a
   <c7>   DW_AT_frame_base  : 0x2c (loc. list)
```

DWARF - Primer

```c
#include <stdio.h>


void do_stuff(int my_arg)
{
   int my_local = my_arg + 2;
   int i;

   for (i = 0; i < my_local; ++i)
      printf("i = %d\n", i);
}


int main()
{
   do_stuff(2);
   return 0;
 }
```

How debuggers work - Eli Bendersky (eli.thegreenplace.net)

```
.debug_info
<2><8a>: Abbrev: 6(DW_TAG_formal_parameter)
   <8b>   DW_AT_name         : (...): my_arg
   <8f>   DW_AT_decl_file    : 1
   <90>   DW_AT_decl_line    : 4
   <91>   DW_AT_type         : <0x4b>
   <95>   DW_AT_location     : (DW_OP_fbreg: 0)
<2><98>: Abbrev: 7 (DW_TAG_variable)
   <99>   DW_AT_name         : (...): my_local
   <9d>   DW_AT_decl_file    : 1
   <9e>   DW_AT_decl_line    : 6
   <9f>   DW_AT_type         : <0x4b>
   <a3>   DW_AT_location     : (DW_OP_fbreg: -20)
<2><a6>: Abbrev: 8 (DW_TAG_variable)
   <a7>   DW_AT_name         : i
   <a9>   DW_AT_decl_file    : 1
   <aa>   DW_AT_decl_line    : 7
   <ab>   DW_AT_type         : <0x4b>
   <af>   DW_AT_location     : (DW_OP_fbreg: -24)
```

DWARF - Primer

```c
#include <stdio.h>


void do_stuff(int my_arg)
{
   int my_local = my_arg + 2;
   int i;

   for (i = 0; i < my_local; ++i)
      printf("i = %d\n", i);
}


int main()
{
   do_stuff(2);
   return 0;
 }
```

How debuggers work - Eli Bendersky (eli.thegreenplace.net)

.debug_loc

```
Offset   Begin    End      Expression
    00000000 08048604 08048605 (DW_OP_breg4: 4 )
    00000000 08048605 08048607 (DW_OP_breg4: 8 )
    00000000 08048607 0804863e (DW_OP_breg5: 8 )
    00000000 <End of list>
    0000002c 0804863e 0804863f (DW_OP_breg4: 4 )
    0000002c 0804863f 08048641 (DW_OP_breg4: 8 )
    0000002c 08048641 0804865a (DW_OP_breg5: 8 )
   0000002c <End of list>
```

.debug_line

File name	Line number	Starting address
tracedprog2.c	5	0x8048604
tracedprog2.c	6	0x804860a
tracedprog2.c	9	0x8048613
tracedprog2.c	10	0x804861c
tracedprog2.c	9	0x8048630
tracedprog2.c	11	0x804863c
tracedprog2.c	15	0x804863e
tracedprog2.c	16	0x8048647
tracedprog2.c	17	0x8048653
tracedprog2.c	18	0x8048658

DA LI OVO MOŽE BRŽE?

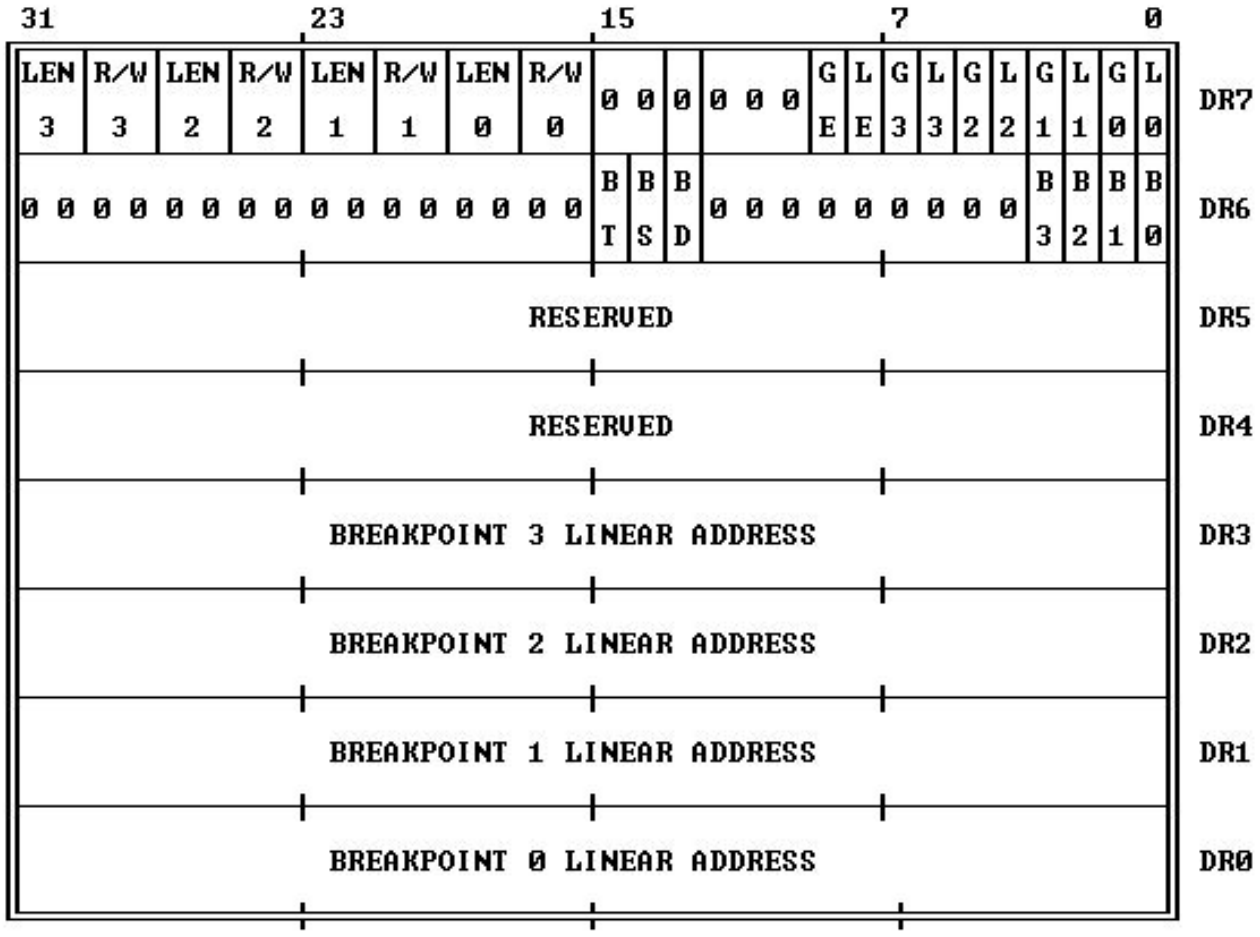
MOŽE

Hardverska podrška

- Neke od funkcionalnosti koje moderni debageri poseduju bi bile jako spore (ili nemoguće) ako bi se oslanjali samo na funkcionalnosti obezbeđene od strane operativnog sistema
- Kako bi ove funkcionalnosti bile ubrzane proizvođači čipova su počeli da ugrađuju poseban hardver koji pomaže sa debagovanjem
- Taj poseban hardver dosta liči na hardverske tačke prekida koje smo videli još na prvim računarima, samo što ih je sada moguće konfigurisati iz softvera bez potrebe za podešavanjem fizičkih prekidača
- Kod Intela hardverska podrška se prvi put pojavljuje sa 80386 procesorom (1985)
- x86 poseduje 6 registrara koji omogućacaju podešavanje do 4 hardverske tačke prekida

Hardverska podrška

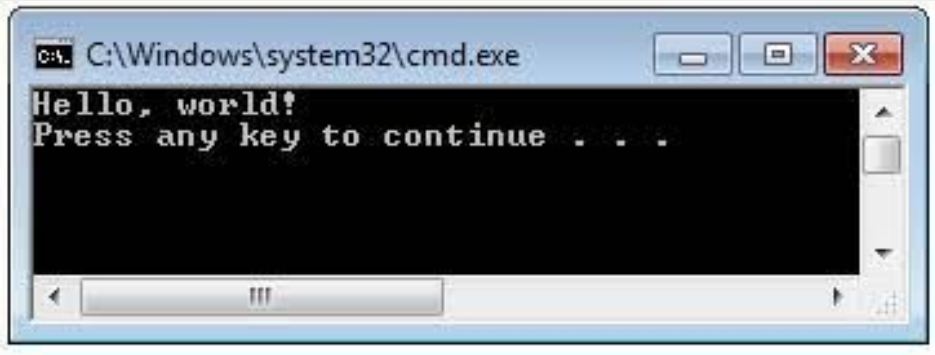
Figure 12-1. Debug Registers

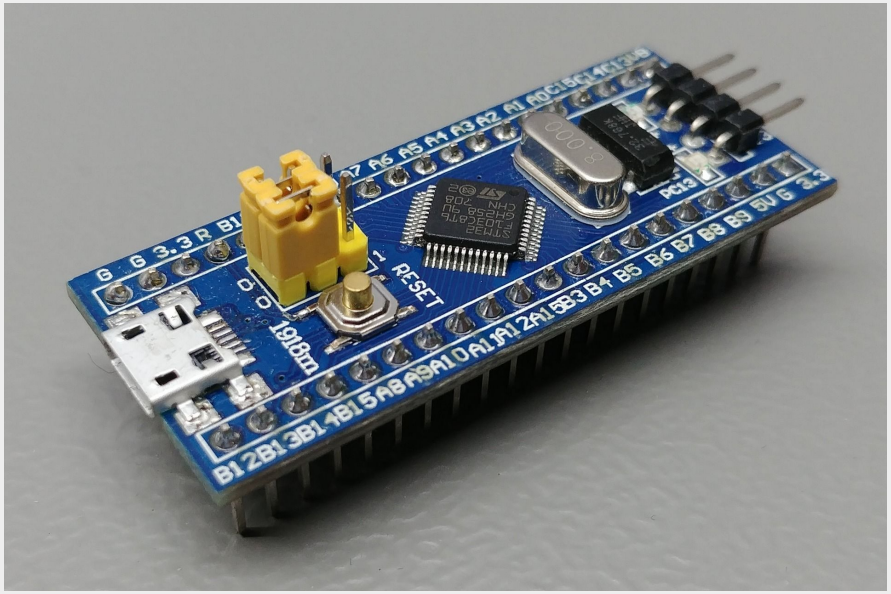


NOTE

0 MEANS INTEL RESERVED. DO NOT DEFINE.

Debagovanje embedded sistema





Debagovanje embedded sistema

- Embedded sistemi često ne poseduju operativni sistem niti način da korisnik interaguje sa debagerom koji bi se pokretao na samom embedded sistemu
- Umesto toga, embedded sistemi imaju opsežnu hardversku podršku za eksterno debagovanje
- Usluge koje obično obavlja operativni sistem su u ovom slučaju u potpunosti implementirane u hardveru
- Taj hardver potom komunicira sa debagerom koji se izvršava na PC računaru uz pomoć nekog komunikacionog protokola

JTAG

- JTAG je komunikacioni protokol koji se najčešće koristi za komunikaciju sa hardverom za podršku debagovanju
-
- Originalno namenjen za *boundary scan*, kasnije proširen da omogući i debagovanje embedded sistema
-
- Fizički sloj standarda koristi 2, 4 ili 5 linija (najčešće su varijante sa 4 linije)
-
- To su:
  - **TDI** (Test Data In)
  - **TDO** (Test Data Out)
  - **TCK** (Test Clock) 10-100 MHz (zavisi od najsporijeg uređaja u lancu)
  - **TMS** (Test Mode Select) prolazi kroz automat stanja JTAG protokola
  - **TRST** (Test Reset) opcion

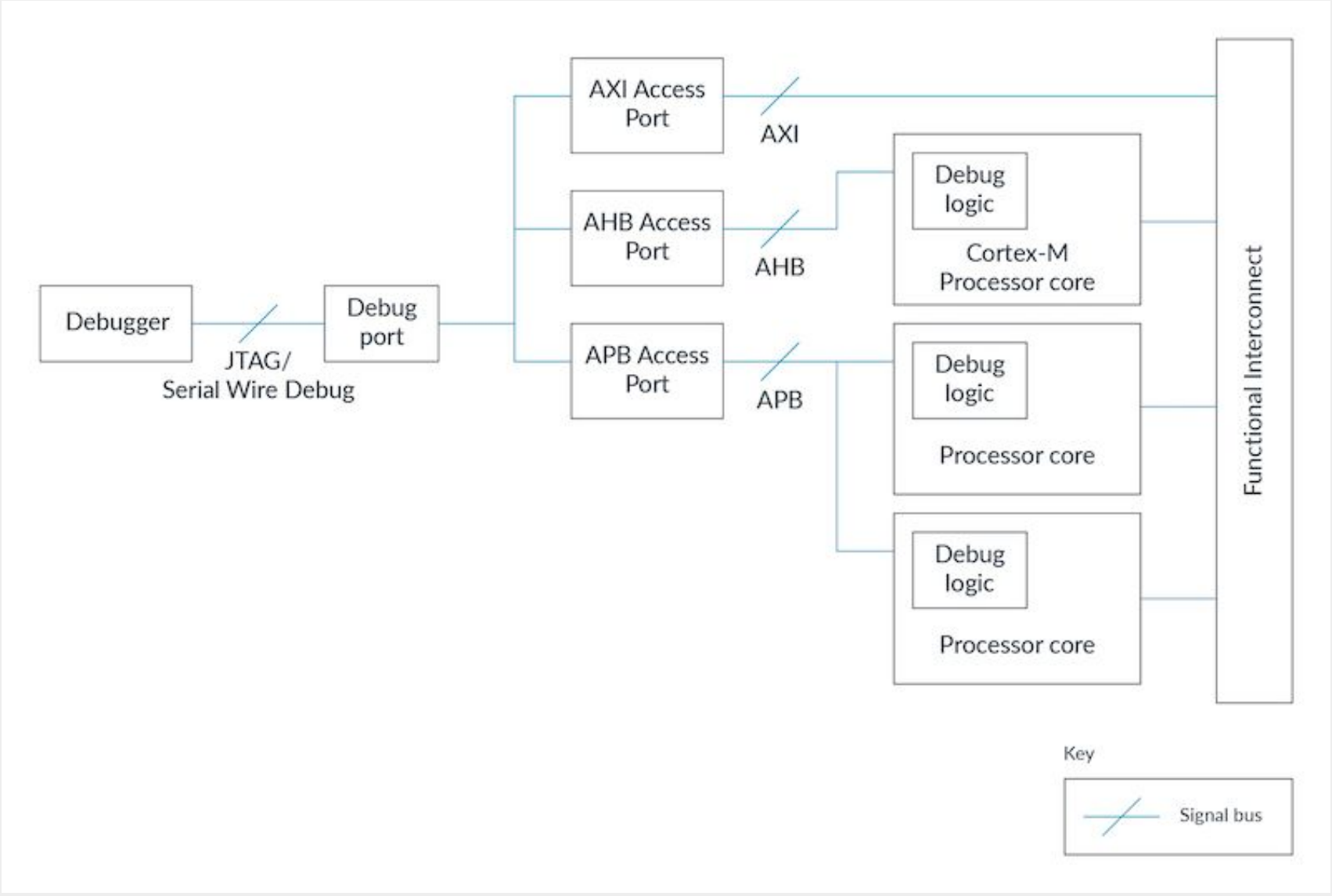
JTAG


LAN
USB
Power
USB
Trace
Target power
Pro
i-Trace
for ARM Cortex-M processors
SEGGER
Target
Target + Trace
Cortex-M Trace Reference Board
freescale

ARM CoreSight

- ARM CoreSight je set tehnologija koje omogućavaju prethodno pomenutu opširnu podršku za eksterno debagovanje
-
- ARM CoreSight obuhvata:
  - Debug Port (Interfejs CoreSight sistema prema eksternom protokolu)
  - Access Port (Interfejs CoreSight sistema prema unutrašnjim magistralama embedded sistema)
  - Core Debug (Interfejs CoreSight sistema prema procesorskom jezgru)
  - System Debug (Set periferija koje omogućavaju napredno debagovanje)
    - Flash Patch and Breakpoint
    - Data Watchpoint and Trace
    - Instrumentation Trace Macrocell
    - Embedded Trace Macrocell
  - Trace Port Interface Unit (Interfejs CoreSight sistema prema prikupljaču tragova)

ARM CoreSight



ARM CoreSight

Table 222. 32-bit debug port registers addressed through the shifted value A[3:2]

Address	A[3:2] value	Description
0x0	00	Reserved, must be kept at reset value.
0x4	01	DP CTRL/STAT register. Used to: – Request a system or debug power-up – Configure the transfer operation for AP accesses – Control the pushed compare and pushed verify operations. – Read some status flags (overrun, power-up acknowledges)
0x8	10	DP SELECT register: Used to select the current access port and the active 4-words register window. – Bits 31:24: APSEL: select the current AP – Bits 23:8: reserved – Bits 7:4: APBANKSEL: select the active 4-words register window on the current AP – Bits 3:0: reserved
0xC	11	DP RDBUFF register: Used to allow the debugger to get the final result after a sequence of operations (without requesting new JTAG-DP operation)

Table 227. Cortex®-M3 AHB-AP registers

Address offset	Register name	Notes
0x00	AHB-AP Control and Status Word	Configures and controls transfers through the AHB interface (size, hprot, status on current transfer, address increment type
0x04	AHB-AP Transfer Address	-
0x0C	AHB-AP Data Read/Write	-
0x10	AHB-AP Banked Data 0	Directly maps the 4 aligned data words without rewriting the Transfer Address register.
0x14	AHB-AP Banked Data 1	
0x18	AHB-AP Banked Data 2	
0x1C	AHB-AP Banked Data 3	
0xF8	AHB-AP Debug ROM Address	Base Address of the debug interface
0xFC	AHB-AP ID register	-

Table 221. JTAG debug port data registers

IR(3:0)	Data register	Details
1111	BYPASS [1 bit]	-
1110	IDCODE [32 bits]	ID CODE 0x3BA00477 (Arm® Cortex®-M3 r1p1-01rel0 ID Code)
1010	DPACC [35 bits]	Debug port access register This initiates a debug port and allows access to a debug port register. – When transferring data IN: Bits 34:3 = DATA[31:0] = 32-bit data to transfer for a write request Bits 2:1 = A[3:2] = 2-bit address of a debug port register. Bit 0 = RnW = Read request (1) or write request (0). – When transferring data OUT: Bits 34:3 = DATA[31:0] = 32-bit data which is read following a read request Bits 2:0 = ACK[2:0] = 3-bit Acknowledge: 010 = OK/FAULT 001 = WAIT OTHER = reserved Refer to *Table 222* for a description of the A[3:2] bits
1011	APACC [35 bits]	Access port access register Initiates an access port and allows access to an access port register. – When transferring data IN: Bits 34:3 = DATA[31:0] = 32-bit data to shift in for a write request Bits 2:1 = A[3:2] = 2-bit address (sub-address AP registers). Bit 0 = RnW= Read request (1) or write request (0). – When transferring data OUT: Bits 34:3 = DATA[31:0] = 32-bit data which is read following a read request Bits 2:0 = ACK[2:0] = 3-bit Acknowledge: 010 = OK/FAULT 001 = WAIT OTHER = reserved There are many AP registers (see AHB-AP) addressed as the combination of: – The shifted value A[3:2] – The current value of the DP SELECT register
1000	ABORT [35 bits]	Abort register – Bits 31:1 = Reserved – Bit 0 = DAPABORT: write 1 to generate a DAP abort.

ARM CoreSight

Table 228. Core debug registers

Register	Description
DHCSR	The 32-bit Debug Halting Control and Status register This provides status information about the state of the processor enable core debug halt and step the processor
DCRSR	The 17-bit Debug Core register Selector register: This selects the processor register to transfer data to or from.
DCRDR	The 32-bit Debug Core register Data register: This holds data for reading and writing registers to and from the processor selected by the DCRSR (Selector) register.
DEMCR	The 32-bit Debug Exception and Monitor Control register: This provides Vector Catching and Debug Monitor Control. This register contains a bit named ***TRCENA*** which enable the use of a TRACE.

Agenda

1. Šta je debager i njihova istorija
2. Kako rade debageri
3. Alternative i saveti

Alternative, ili ne baš...

- Post-mortem debugging
- Remote debugging
- Logging
- Tracing
- Breakless debugging
- Record and replay debugging
- Hot reload
- Time travel debugging
- Profiling
- Instrumentation
- Instruction set simulator

Saveti

- GDB WinDbg
- Watchpoints
- Expression evaluation
- Memory view

Linkovi

- Playing with ptrace - Pradeep Padala (www.linuxjournal.com/article/6100)
- How debuggers work - Eli Bendersky (eli.thegreenplace.net/2011/01/23/how-debuggers-work-part-1)
- How debugger works - Alex on Linux (www.alexonlinux.com/how-debugger-works)
- The Debugging Application Programming Interface - Microsoft Learn (learn.microsoft.com/en-us/previous-versions/ms809754(v=msdn.10))
- Introduction to the DWARF Debugging Format - Michael J. Eager (dwarfstd.org/doc/Debugging using DWARF-2012.pdf)
- Debug Interface Access SDK - Microsoft Learn (learn.microsoft.com/en-us/previous-versions/visualstudio/visual-studio-2015/debugger/debug-interface-access/debug-interface-access-sdk)
- x86 Debug Registers - Intel 80386 Reference Programmer's Manual (pdos.csail.mit.edu/6.828/2004/readings/i386/s12_02)
- What a good debugger can do - Andy Hippo (werat.dev/blog/what-a-good-debugger-can-do)
- Non-Breaking Breakpoints: The Evolution Of Debugging - Noa Goldman (www.rookout.com/blog/non-breaking-breakpoints-the-evolution-of-debugging)
-

Linkovi pt.2

- What is JTAG and how can I make use of it? - XJTAG (www.xjtag.com/about-jtag/what-is-jtag)
- Technical Guide to JTAG - XJTAG (www.xjtag.com/about-jtag/jtag-a-technical-overview)
- Learn the architecture - Introducing CoreSight debug and trace - arm Developer (developer.arm.com/documentation/102520/0100)
- Debug support - Cortex-M3 Technical Reference Manual - arm Developer (developer.arm.com/documentation/ddi0337/e/Core-Debug)
- How to debug embedded systems - Ilias Alexopoulos (www.edn.com/how-to-debug-embedded-systems)
- Debugging with Cortex-M3 Microcontrollers - embedded.com (www.embedded.com/debugging-with-cortex-m3-microcontrollers)
- Small matter of programming

Hvala na pažnji!

Pitanja?

Instrukcije

Kopirajte prezentaciju pa editujte, **ne menjajte** direktno ovu!
Staviti gotovu prezentaciju u folder prezentacije, da ljudi odrade peer-review

- **Ne menjati:**
  - Font i size
  - Izgled slajdova, poziciju stvari na njima
- **Prilagoditi:**
  - Prvi slajd (tema, predavač, datum)
  - Agendu (ako želite da je imate)

  - Footer (View → Team Builder → Promena *predavača i teme* na oba slajda u Layouts)