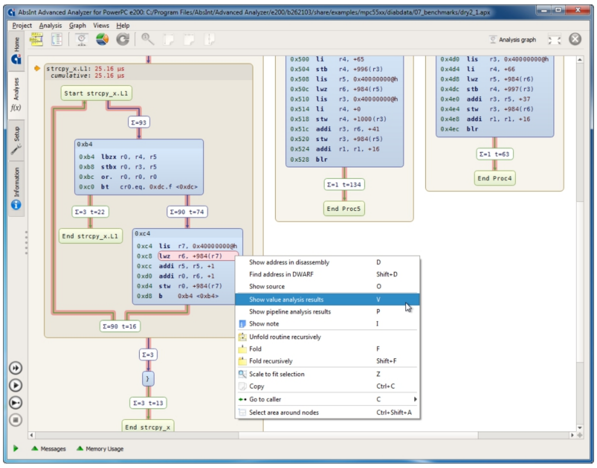
ValueAnalyzer

Static Value Analysis for Embedded Systems

ValueAnalyzer is a tool for static analysis of register and memory contents. It is particularly useful for:

- verifying the absence of illegal accesses to an embedded software system from within third-party modules provided as object code,
- initialization analysis for execution-time or stack-usage analyses, to improve precision and minimize manual annotations.



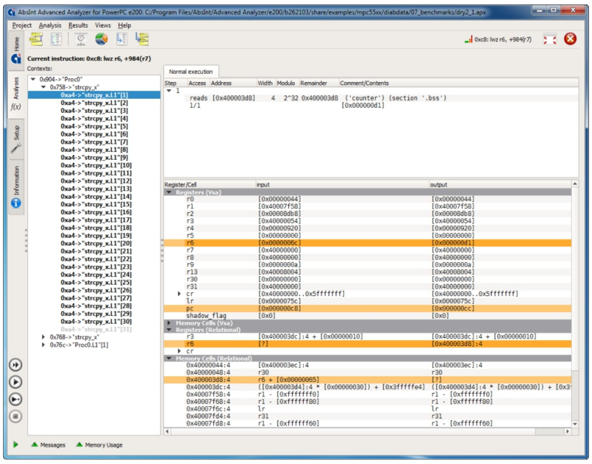
```
0xb4
0xb4  lbzx r0, r4, r5
0xb8  stbx r0, r3, r5
0xbc  or.  r0, r0, r0
0xc0  bt   cr0.eq, 0xdc.f <0xdc>
```

```
0xc4
0xc4  lis   r7, 0x40000000@h
0xc8  lwz   r6, +984(r7)
0xcc  addi  r5, r5, +1
0xd0  addi  r0, r6, +1
0xd4  stw   r0, +984(r7)
0xd8  b     0xb4 <0xb4>
```

```
0x500  li    r4, +65
0x504  stb   r4, +996(r3)
0x508  lis   r5, 0x40000000@h
0x50c  lwz   r6, +984(r5)
0x510  lis   r3, 0x40000000@h
0x514  li    r4, +0
0x518  stw   r4, +1000(r3)
0x51c  addi  r3, r6, +41
0x520  stw   r3, +984(r5)
0x524  addi  r1, r1, +16
0x528  blr
```

```
0x4d0  lis   r3, 0x40000000@h
0x4d4  li    r4, +66
0x4d8  lwz   r5, +984(r6)
0x4dc  stb   r4, +997(r3)
0x4e0  addi  r3, r5, +37
0x4e4  stw   r3, +984(r6)
0x4e8  addi  r1, r1, +16
0x4ec  blr
```

Visualization of Control Flow Graph...



AbsInt Advanced Analyzer for PowerPC e200: C:/Program Files/AbsInt/Advanced Analyzer/e200/b262103/share/examples/mpc55xx/diabdata/07_benchmarks/dry2_1.apx
Project
Analysis
Results
Views
Help
0xc8: lwz r6, +984(r7)
Home
Analyses
Setup
Information
Normal execution
Messages
Memory Usage

Current instruction: 0xc8: lwz r6, +984(r7)

Contexts:

```
▼ 0x904->"Proc0"
  ▼ 0x758->"strcpy_x"
      0xa4->"strcpy_x.L1"[1]
      0xa4->"strcpy_x.L1"[2]
      0xa4->"strcpy_x.L1"[3]
      0xa4->"strcpy_x.L1"[4]
      0xa4->"strcpy_x.L1"[5]
      0xa4->"strcpy_x.L1"[6]
      0xa4->"strcpy_x.L1"[7]
      0xa4->"strcpy_x.L1"[8]
      0xa4->"strcpy_x.L1"[9]
      0xa4->"strcpy_x.L1"[10]
      0xa4->"strcpy_x.L1"[11]
      0xa4->"strcpy_x.L1"[12]
      0xa4->"strcpy_x.L1"[13]
      0xa4->"strcpy_x.L1"[14]
      0xa4->"strcpy_x.L1"[15]
      0xa4->"strcpy_x.L1"[16]
      0xa4->"strcpy_x.L1"[17]
      0xa4->"strcpy_x.L1"[18]
      0xa4->"strcpy_x.L1"[19]
      0xa4->"strcpy_x.L1"[20]
      0xa4->"strcpy_x.L1"[21]
      0xa4->"strcpy_x.L1"[22]
      0xa4->"strcpy_x.L1"[23]
      0xa4->"strcpy_x.L1"[24]
      0xa4->"strcpy_x.L1"[25]
      0xa4->"strcpy_x.L1"[26]
      0xa4->"strcpy_x.L1"[27]
      0xa4->"strcpy_x.L1"[28]
      0xa4->"strcpy_x.L1"[29]
      0xa4->"strcpy_x.L1"[30]
      0xa4->"strcpy_x.L1"[31]
  ▶ 0x768->"strcpy_x"
  ▶ 0x76c->"Proc0.L1"[1]
```

```
Step  Access  Address       Width  Modulo  Remainder   Comment/Contents
▼ 1
      reads   [0x400003d8]    4    2^32    0x400003d8  ('counter') (section '.bss')
      1/1                                              [0x000000d1]
```

```
Register/Cell       input                                            output
▼ Registers (Vsa)
  r0                [0x00000044]                                     [0x00000044]
  r1                [0x40007f58]                                     [0x40007f58]
  r2                [0x00008db8]                                     [0x00008db8]
  r3                [0x40000054]                                     [0x40000054]
  r4                [0x00000920]                                     [0x00000920]
  r5                [0x00000000]                                     [0x00000000]
  r6                [0x0000006c]                                     [0x000000d1]
  r7                [0x40000000]                                     [0x40000000]
  r8                [0x40000000]                                     [0x40000000]
  r9                [0x0000000a]                                     [0x0000000a]
  r13               [0x40008004]                                     [0x40008004]
  r30               [0x00000000]                                     [0x00000000]
  r31               [0x40000000]                                     [0x40000000]
▶ cr                [0x40000000..0x5fffffff]                         [0x40000000..0x5fffffff]
  lr                [0x0000075c]                                     [0x0000075c]
  pc                [0x000000c8]                                     [0x000000cc]
  shadow_flag       [0x0]                                            [0x0]
▶ Memory Cells (Vsa)
▼ Registers (Relational)
  r3                [0x400003dc]:4 + [0x00000010]                    [0x400003dc]:4 + [0x00000010]
  r6                [?]                                              [0x400003d8]:4
▶ cr
▼ Memory Cells (Relational)
  0x40000044:4      [0x400003ec]:4                                   [0x400003ec]:4
  0x40000048:4      r30                                              r30
  0x400003d8:4      r6 + [0x00000065]                                [?]
  0x400003dc:4      ([0x400003d4]:4 * [0x00000030]) + [0x3ffffe4]    ([0x400003d4]:4 * [0x00000030]) + [0x3
  0x40007f58:4      r1 - [0xfffffff0]                                r1 - [0xfffffff0]
  0x40007f68:4      r1 - [0xffffff80]                                r1 - [0xffffff80]
  0x40007f6c:4      lr                                               lr
  0x40007fd4:4      r31                                              r31
  0x40007fd8:4      r1 - [0xffffff60]                                r1 - [0xffffff60]
```

... with value information for selected instruction.

Why do you need ValueAnalyzer?

- ValueAnalyzer is an **add-on** to aiT, StackAnalyzer, TimingProfiler, and TimeWeaver.
- In embedded control software global data structures like (function) pointer variables are often initialized by separate initialization tasks. When a periodic or asynchronous task or runnable is analyzed for its worst-case timing or stack usage, such initializations are unknown.
  With ValueAnalyzer you can **automatically collect initialization data** for global variables and feed them as auto-generated annotations to a WCET, stack-usage, or another value analysis. Hence, it can **improve the precision** of aiT/StackAnalyzer/TimingProfiler/TimeWeaver analyses and **avoid manual annotations**.
- ValueAnalyzer helps to **detect illegal memory accesses** caused by programming errors, non-compliance with system specifications, or compiler/linker errors. It can verify **static user-defined assertions**, such as
  - assertions that a user-selected instruction or function does (not) access a user-selected memory area, or
  - global assertions on user-selected memory areas (is read-only, is write-only, is inaccessible, ...).
- ValueAnalyzer visualizes the call and control flow graph and allows to easily explore the analysis results. The contents of registers and memory cells are shown before and after each instruction execution context.
- ValueAnalyzer creates comprehensive reports for documentation and certification purposes, including:
  - all memory accesses per function
  - all calls to user-selected functions
  - all accesses to user-selected memory areas, register and memory contents at entry/exit of selected functions.
- ValueAnalyzer supports batch mode execution and integration in **continuous test and integration** frameworks.

Supported processors and compilers

ValueAnalyzer is available all processor–compiler combinations supported by aiT, StackAnalyzer, TimingProfiler, and TimeWeaver.