It’s a Bug, not a Feature: Pitfalls of initialized Block RAM use in soft processor systems

Filip Brkic
Dept. of Information Technology
Mannheim University of Applied Sciences
Mannheim, Germany
filip.brkic@stud.hs-mannheim.de

Ruediger Willenberg
Dept. of Information Technology
Mannheim University of Applied Sciences
Mannheim, Germany
r.willenberg@hs-mannheim.de

***Abstract*—Unlike SRAM in most common embedded systems, dedicated RAM blocks on configured FPGAs are already initialized when booting on-chip embedded systems. This potential benefit can lead to software malfunction if used improperly.**

We identify how problematic design choices in the AMD/Xilinx toolflow for the Microblaze soft processor can lead to corruption of initial program state. We suggest fixes to prevent this flawed behaviour. Furthermore, we survey other FPGA vendors’ soft processor toolchains to probe for similar problems.

***Index Terms*—FPGA, soft CPU, embedded, SRAM, BlockRAM**

I. Introduction

Soft processor systems are a useful addition to many FPGA designs. Compute, control and communications tasks that don’t require high throughput can be implemented, verified and modified more quickly and easily in software, saving design time and, potentially, area resources, compared to task-specific hardware. Because soft processor and peripheral IP are implemented in regular logic fabric like any other digital circuit, they require no special FPGA features beside sufficient area.

Virtually any modern FPGA also comes with some form of block *Random Access Memory* (RAM), which provides higher memory density for storage purposes than LUTs or flip-flops do. In an FPGA design, block RAMs combine some properties of standard *Read-Only Memory* (ROM) and RAM: They are initialized as part of the configuration process, emulating the persistence of ROM. At the same time, as RAM they are mutable and their content can be overwritten during operation.

These properties suggest an advantage for soft processor systems: System memory can already be loaded with application code and initialized variables when FPGA operation commences after configuration. However, a downside appears if the processor system is restartable without reconfiguring the FPGA: Global variables might have been modified, and even code in RAM could have been accidentally or maliciously changed.

The rest of this paper is organized as follows: In Section II, we are demonstrating how software malfunction can occur when these block RAM properties are used improperly. We illustrate in Section III why this happens by comparing the use of memories in FPGA and non-FPGA embedded systems. In Section IV, we discuss the implications for stability and security and take up possible objections to our case. Section V presents several workarounds for the AMD Microblaze toolchain to avoid malfunction. We analyze other FPGA vendors’ toolchains in Section VI and compare if the same issues can occur. Section VII concludes.

II. Example malfunction

To illustrate how simple it is to trigger software malfunction, we use the *AMD* (formerly Xilinx) FPGA embedded toolchain for the *Microblaze* processor (Table I).

TABLE I
AMD EMBEDDED TOOLCHAIN

Core	Microblaze 11.0
HW tools	Vivado / IP Integrator 2023.1
SW tools	Vitis 2023.1
Board	Digilent Zybo Z7-20
FPGA	Zynq XC7Z020CLG400-1

Note: With the 2024.1 release, AMD has introduced the *Microblaze-V*, a RISC-V core with the same hardware connectivity as the legacy Microblaze. Furthermore, they have deprecated the *Eclipse*-based Vitis IDE and replaced it with a *VS Code/Theia*-based Vitis IDE.
We have established that the same issues examined here persist, and our code repository introduced in Section V provides workarounds compatible with both cores and both IDEs.

All of the five vendor toolchains we examined share the following capabilities:

- A complete source-to-bitstream FPGA flow
- A design tool to create embedded systems at the component and bus connectivity level
- A software IDE to design, debug and compile applications for the soft processor
- Automatic generation of a *Board Support Package* (BSP) based on imported properties of the processor system: A *standalone* (*“bare-metal”*) runtime environment with drivers that abstract peripheral access, and a linker script to build a complete C/C++ application (the exception is

Microchip, who offer template BSPs and applications that require editing to fit the hardware system's properties).
- Compiled applications can be inserted into an FPGA bitstream so that application code and data are stored as initial state in block RAMs

We equip our minimal system with 16 KBytes of RAM used for both instruction and data storage, a UART for standard output, a reset manager and a clock manager. The reset manager can trigger a *soft reset*: A reboot of the processor without reconfiguration of the entire FPGA, and without a hardware reset of memory state. *Soft reset* can be triggered by an external reset signal or on loss of clock frequency lock.

We write the following simple C program:[1]

```
#include <stdio.h>
#include <stdlib.h>

int  globalint = 42, *dynarray = NULL;

int main()
{
   xil_printf("globalint: %d\n", globalint);
   globalint++;

   dynarray = malloc(100 * sizeof(int));
   xil_printf("Dynamic array address: ");
   xil_printf("0x%p\n\n", dynarray);

   return 0;
}
```

After compilation, linking and bitstream integration, we download the bitstream to the FPGA and monitor the standard output via UART. We then use a board-level signal to trigger a *soft reset* twice, resulting in the following output:

```
globalint: 42
Dynamic array address: 0x2eb8

  -- first soft reset --
globalint: 43
Dynamic array address: 0x0

  -- second soft reset --
globalint: 44
Dynamic array address: 0x0
```

On the first execution after configuration, the program behaves as expected: The init value `42` of the variable `globalint` is printed out in decimal format. The value of `globalint` is then incremented by one, but the altered value is not used or printed out. After a request of dynamic memory allocation via `malloc()`, the returned address is printed out in hexadecimal format.

After *soft reset*, the program behaves differently: The variable `globalint` is still in its incremented state from the previous execution run. The `malloc()`-Function fails to allocate the requested dynamic memory, signaling this with the return of a NULL address.

This behaviour is not conforming to the ISO C standard [1], which clearly states that *"All objects with static storage duration shall be initialized (set to their initial values) before program startup."*[2].

III. Background

To explain the nonconforming behaviour, we will briefly recall common knowledge about program memory structure. For all further discussion, we assume single-application, *bare-metal* designs with only on-chip memory. Larger designs running on operating systems and with multiple threads or tasks, often supported by off-chip DRAM, use different bootloading mechanisms for their applications, although their post-reset bootloaders are potentially vulnerable to the same issues.

A. ELF sections and memory segments

All soft processor toolchains discussed here use variations of the *GNU Compiler Collection* [2] that produce ELF [3] binaries after compilation and linking. These binaries commonly include at least the following (or similarly named) file *sections*, which specify the content and layout of memory *segments* in the target system:

- **.text**: The executable instruction code.
- **.rodata**: Constant data not intended to be changed (indicated by the *const* keyword in C).
- **.bss**: Global (static) variables and structures that are initialized to 0. Only the RAM location and segment size are stored in the ELF file, the runtime library's code initializes it before *program startup*.
- **.data**: Global (static) variables and structures that are initialized to a nonzero value before *program startup*.
- **.heap**: The location and size of memory for dynamic allocation. e.g. with `malloc()`. No initialization is required before *program startup*.
- **.stack**: The location and size of memory for local (automatic) variables and function call information. Stack data ist automatically managed during runtime, no initialization is required.

B. Memory and startup in common embedded systems

Common embedded systems use a combination of ROM (mostly Flash memory) and RAM: ROM is persistent without power and immutable by regular store instructions; RAM is mutable, but uninitialized at power-up. The ELF file sections are distributed into ROM and RAM as illustrated in Figure 1: The **.text** and **.rodata** are located in ROM, which also holds a constant copy of the **.data** segment (also known as a *load segment)*. The boot code in ROM is responsible for segmenting and initializing the RAM according to the section information from the ELF file. The **.stack** and **.heap** segments

[1]Experienced embedded programmers will already be tempted to point out that neither the use of static (global) variables nor the use of dynamic memory in low-resource embedded systems is recommended; we will discuss this in Section IV.

[2]In this context, *program startup* refers to the entry into the `main()` function; it is also common to name the C runtime library code that initializes a system as *startup code*. To avoid confusion, we will use the term *boot code* in this paper for the latter.

stay uninitialized, but objects like the *stack pointer* will be initialized with their boundaries.

Before the `main()` function can be called, the entire **.bss** needs to be set to zero values and the **.data** segment needs to be initialized from its immutable copy in ROM, the *load segment*, so that all global (static) variables are in their initial state as the C standard dictates.

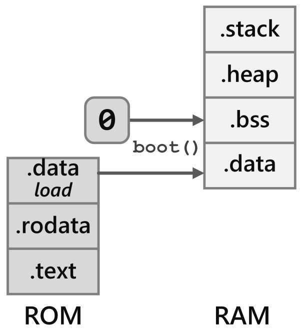

Fig. 1. Memory segments in an embedded system with ROM and RAM

C. *Memory and startup in* Microblaze *embedded systems*

As FPGA memory blocks combine ROM and RAM properties in a convenient way, the *Vitis* tool flow places all ELF file sections into their intended memory segments in an initialized state when adding a *bare-metal* application to the bitstream. From startup, there is only one contiguous stretch of memory as illustrated in Figure 2. The linker script automatically generated for the application by *Vitis* does not provide for extra load segments for the global data. Consequently, the boot code that is linked into the C application fills the **.bss** segment with zeroes, but it does not initialize **.data** from a load segment.

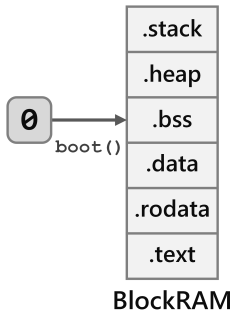

Fig. 2. Default assignment of memory segments into Block RAM by *Vitis*

This is the cause of the problematic *soft reset* behaviour demonstrated above. Any global variables that are modified during program execution will still be in the same state after reset. The *newlib* [4] implementation of the C standard library used by *Vitis* is also dependent on initialized global data tables, therefore `malloc()` fails on a second run.

IV. Discussion

Strictly speaking, any software package for C development that results in non-standard-conforming behaviour is problematic, and this seems to be a long-standing issue. The assembler boot code in `crtinit.S` that lacks initialization of the **.data** segment has not been changed since 2009, according to copyright and versioning data [5].

There are a number of reasonable objections that can be made to our criticism, which we will briefly discuss here.

A. *Objections*

1) "Use of global variables should be avoided anyway. Dynamic memory allocation should not be used in memory-limited embedded systems anyway.":

These are reasonable recommendations regarding C coding style, true in most cases for the former statement, reasonable with regard to resource-limited systems for the latter.

However, any formally correct use of C language elements should work as set out in the language standard. It is up to a properly skilled programmer to make design choices beyond that.

2) "This is well known by experienced embedded programmers and can be worked around.":

This argument reflects that there is a reasonable higher expectation on embedded programmers to be aware of implementation details beyond formal language, and that these programmers are used to dealing with imperfect implementations due to limited resources. The authors are not claiming to have identified this issue for the first time; rather, we write this to create such awareness for this specific problem in context of FPGA block RAM. We also offer such workarounds in Section V.

However, with their generally well-designed tool integration, AMD is kindly offering a smooth path from hardware system creation straight to functioning embedded applications. These programs should, by default, work under all reasonable circumstances.

3) "Including a load segment for data and extra initialization code wastes memory.":

While two segments of initialized globals and the extra boot code might indeed waste limited block RAM space, it would be easy to make both features *optional* in the *Board Support Package* settings (although we would strongly recommend making the "safe" option the default). As we observe in Section VI, this is exactly what *Altera* offers in their embedded flow.

4) "This is only a problem if there is a reset signal in the system.":

Aside from an external reset signal that might indeed not be included in some systems, the soft reset mechanism is also available for debug resets, watchdog timeouts and loss of clock frequency lock. All of these are important features to design and run stable embedded designs.

5) "Can't we just use the FPGA's Global Set/Reset *(GSR) to re-initialize Block RAM content?":*

According to AMD/Xilinx, while the overall initial device state can be restored from the configuration bitstream with the *Global Set/Reset* signal, only a Block RAM's output latches are reset by the GSR, not its internal memory state [6].

B. Security implications and lack of real ROM

Buffer overflows exemplify that any software stability issue, especially one leading to memory corruption, is also a potential security problem. If the affected system is directly or indirectly connected to a network, more extensive security issues may result.

As discussed, even a correct C program can lead to malfunction after a soft reset if the runtime environment leaves the door open to malfunction. In Section V, we offer a minimum fix to prevent the *.data* segment issue described.

However, additional harm is possible with unintentional coding errors or even maliciously inserted code: If block RAM data remains in an altered state after reset, even instruction code, declared constants or load segments could be corrupted. This issue does not affect common embedded systems where these segments are stored in read-only memory, but is common to all FPGA embedded systems that put all their instruction and data in RAM. Therefore, we also offer a hardware solution to make a part of a Microblaze system's block memory read-only.

V. WORKAROUNDS FOR THE MICROBLAZE(-V)

In our linked Github repository [7] we provide code for several options to fix the identified issues for AMD Microblaze and Microblaze-V systems. Code is offered under the liberal MIT license. It should be possible to adapt code to other vendors' toolchains surveyed in Section VI.

A. Linker Script Option 1: Adding load segments to a single-memory linker script

To be able to restore initialized globals after reset, extra load segments needs to be defined for each affected section. This data then needs to be copied over to the actual segment at boot. In the default linker scripts auto-generated by Vitis, the sections **.data**, **.sdata** (small globals reachable by 16-bit pointer offset) and **.tdata** (thread-owned global data for small RTOSes) need to be designated this way (**.sdata2** is a misnomer as it holds small *constants*).

The GCC linker's **`AT`** keyword enables specifying this copy and identifying the copy's address. The same mechanism is used in embedded systems with a regular ROM/RAM split.

Our provided bash script **add_load_segments.sh** can be used to automatically extend the *Vitis*-generated linker script with the required modifications:

```
source add_load_segments.sh
  APPSRC_PATH/linker.ld
```

The script modifies **linker.ld** in-place and makes an unchanged backup copy.

Figure 3 illustrates the modified memory layout for the **.data** segment and how it is initialized at boot. To implement the boot copy, the modifications in Subsections V-C or V-D need to be added to the application.

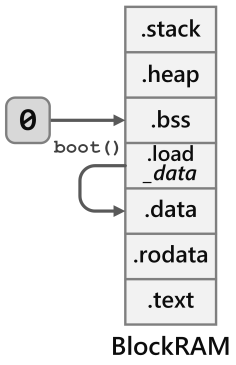

Fig. 3. Memory layout with added load .data section

The following code snippet illustrates how our shell script extends the **.data** section of the linker script and how this could be applied to other vendors' linker scripts with initialized data sections. The **`bold`** code is the added code.

```
.data : AT ( __load_data_start) {
   . = ALIGN(4);
   __data_start = .;
   *(.data)
   *(.data.*)
   *(.gnu.linkonce.d.*)
   __data_end = .;
} > microblaze_0_local_memory

.load_data (NOLOAD) : {
   . = ALIGN(4);
   __load_data_start = .;
   . += SIZEOF(.data);
} > microblaze_0_local_memory
```

The **`AT`** keyword specifies that while memory space needs to be reserved here for the **.data** segment, its actual content should go do a different memory address, **`__load_data_start`**. A second section called **.load_data** is declared, and at its beginning that address **`__load_data_start`** is declared. While the actual initial data values will go here, the memory space for this new segment still has to be explicitly allocated; this is accomplished by the **`. += SIZEOF(.data);`** statement, which refers to the size of the **.data** segment above. The confusingly named **`(NOLOAD)`** attribute indicates that this added segment is a *source* for the boot loading, not a *target*.

B. Linker Script Option 2: Splitting the memory into ROM and RAM sections

A second way to change the linker script is to split the single block RAM memory range into two logical segments representing a ROM and a RAM. Sections like **.text** and **.rodata** will be placed in the ROM section, as will load segments for the data sections. The intention behind this approach is to then physically modify the system so that the memory portion labeled "ROM" can actually not be overwritten anymore, therefore protecting the code and data stored there. This requires hardware modifications described in Subsection V-E.

Figure 4 illustrates the changed memory layout, which now places the load copy of the **.data** segment in the ROM-declared range ahead of the target segment. A boot copy according to Subsections V-C or V-D is still required.

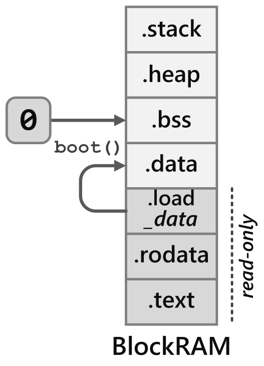

Fig. 4. Memory layout with write-protected sections

To modify the existing linker script correspondingly also involves calling a provided bash script:

```
source split_into_rom_ram.sh \
  --ramaddr=0x3000 APPSRC_PATH/linker.ld
```

The *ramaddr* parameter specifies where the writeable block RAM segment begins; all addresses below belong to the "ROM" segment. Sections are assigned accordingly. Again, the `AT` keyword is used for data sections that require a load segment in the ROM range.

The downside to this approach is that splitting the existing physical block RAM into two halves can be wasteful. Either block RAM is generously alloted in the hardware design and then split with room to spare into the two parts, or an undersized half leads to linker errors. The start address of the writable RAM does then need to be adjusted in the linker script. Furthermore, the corresponding write protection parameter in the hardware design also needs to be changed, and therefore requires rebuilding the bitstream.

The following code snippets illustrate how our shell script institutes the ROM/RAM split in the linker script, and how this could be applied to other vendor's linker scripts. The **`bold`** code is the added or substituted code.

The originals script for a system with 128 KBytes of Block RAM might define its memory like this:

```
MEMORY
{
  mb_memory : ORIGIN = 0x50, LENGTH = 0x1FFB0
}
```

With a specified start of RAM at **`0x3000`**, the modified memory definition will look like this:

```
MEMORY
{
  mbROM : ORIGIN = 0x50, LENGTH = 0x2fb0
  mbRAM : ORIGIN = 0x3000, LENGTH = 0x1d000
}
```

Note: The declared memory only starts at **`0x50`** because the space below is reserved for the Microblaze's reset and interrupt vectors, which the linker script sets with explicit address specification.

The modified script now has to differentiate between three types of sections: Immutable sections like **.text** and **.rodata** get assigned to **`>mbROM`**, while uninitialized variable memory like **.bss**, **.stack** and **.heap** get assigned to **`>mbRAM`**. The changes to an initialized global section like **.data** look like this:

```
.data : {
   . = ALIGN(4);
   __load_data_start = LOADADDR(.data);
   __data_start = .;
   *(.data)
   *(.data.*)
   *(.gnu.linkonce.d.*)
   __data_end = .;
} > mbRAM AT > mbROM
```

Here we can use the **`AT`** keyword in a more compact way: The section assignment in the last line says allocates the space both for **.data** in **`mbRAM`** and for our load segment in **`mbROM`**, the latter of which will be filled with the actual initial values. In the second line, we again add a definition for the pointer **`__load_data_start`**. While the definition is located in the **.data** section, the **`LOADADDR`** command makes it point towards the start of the load section in **`mbROM`**. The boot code can use it as a source address for the init copy.

C. Boot Code Option 1: Load copy with C header and function call

The simplest way to add initialization of global data from their load segments is with copy loops at the beginning of an application's `main()` function. The required target and load addresses for **.data**, **.sdata**, **.tdata** are exported as symbols by the linker and can be stored in pointers. We have packaged the symbol linkage and copy function in a header file that can be added to the source folder and then included with

```
#include "init_mb_globals.h"
```

The first statement in `main()` should then be the function call

```
init_mb_globals();
```

D. Boot Code Option 2: Load copy with linked library

Alternatively, the same C function can be packaged as the initialization function (“constructor”) of a static library; this library is then linked to the application. The C runtime’s startup code calls all constructors before `main()` is called, therefore the segment copies are executed in time. There is no notable overhead in binary size compared to adding the copy function in the C source, but this way, the application code does not need to be modified.

We provide the source file and a bash script to compile and package it in a static library with the Microblaze/RISC-V GCC tools. The script also provides the exact option syntax that needs to be added to the linker call.

E. Adding write protection to the ROM range

Unfortunately, the proprietary design of the Microblaze *Local Memory Bus* (LMB) block does not accommodate separate ROM and RAM blocks in the hardware design - the tool warns that adding data to the bitstream is only possible for one target block per processor. Another possible solution based on existing hardware would be use of the Microblaze’s *Memory Protection Unit*, a limited version of the MMU required for virtual memory and paging. Even without full virtual memory support, this involves a disproportionate amount of extra hardware and software resources.

As an alternative, we have designed a crude but effective solution to make part of the systems block RAM read-only: A configurable VHDL component disables the *Write Enable* signals of the Data LMB if the targeted address is below the configured *writable* range. The range below is therefore protected and acts like a ROM.

To implement this, the local memory block attached to the Microblaze has to be ungrouped and the VHDL component has to be inserted into the Data LMB’s address and write enable lines, as depicted in Figure 5.

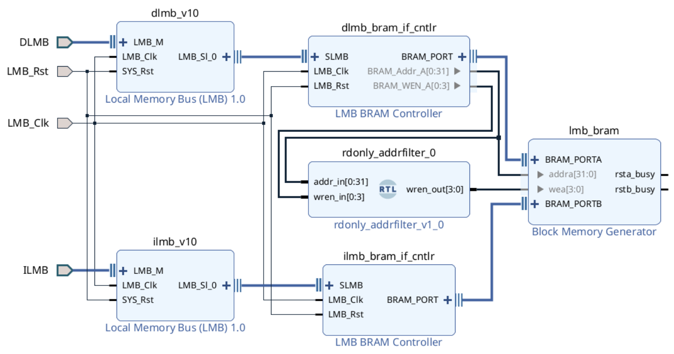

Fig. 5. Microblaze LMB block with added write-protect

F. Memory / Area / Performance overhead of the workarounds

To examine the impact of our workarounds on system resources and performance, we built a minimal Microblaze system with 128 KByte on our *Digilent Zybo Z7-20* board with a *Zynq XC7Z020CLG400-1* FPGA, using Vivado 2023.1. The implemented system requires 865 Flipflops, 1100 LUTs and meets timing up to a system clock speed of 166 Mhz.

The linker script modifications only impact the amount of memory required by a compiled application, not other hardware resources or timing. Inherent to the approach of having a loadable copy of each global data segments, the required amount of memory for global, initialized data doubles. Even a minimal C program with an empty `main()` function requires 252 Bytes for the standard runtime library, which doubles to 504 Bytes with the load segments. Using the `malloc()` function adds another 1040 Bytes in an unchanged setup, therefore 2080 Bytes with the load copy.

Splitting the memory into *read-only* and *writeable* sections has the potential to waste additional memory space, unless the split address is finely adjusted to the required amount of space for ROM segments after building the software. As the intent is to use hardware support to enforce the ROM behaviour, this code-dependent start address then will have to be entered into the hardware design, and the design will have to be re-synthesized and implemented at least in part. Besides the extra implementation time, a fine-grained split address instead of round binary number will add some area and timing impact (see below).

The added startup code to load the global data segments will increase the required code memory. Compiled with *mb-gcc* with the default *-O0* optimization, the statically linked library solution (no change to `main()`) adds 280 Bytes to the **.text** section, while including and calling it as a function would add 304 Bytes. Higher compiler optimization levels do not have an impact. We did not measure the added execution time for the load copy, but using reasonable assumptions this should remain in the sub-millisecond range.

Adding the configurable write protection to the Microblaze’s memory block adds area and timing penalties that depend on the exact address being checked against. Using the round hexadecimal number `0x2000` as an example, only **4** Look-Up-Tables are added to the previous 1100 LUTs (less than +0.4%). Since Block RAM access from the processor’s pipeline constitutes the system’s critical path, the maximum system frequency drops from 166 to 161 MHz (-3%). The area and timing penalties might increase if more non-zero address bits have to be compared exactly.

VI. Other vendors/toolchains

A. Altera (former Intel PSG)

Altera is offering both the legacy *Nios II* soft processor and the *RISC-V* [8]-compatible *Nios V*, both with similar memory, interconnect and peripherals. Generating BSP code and linker scripts from processor system properties is fully automated as with AMD tools. By default, both load data segments and boot copy code are included in the runtime library. However, Altera’s *BSP Editor* gives programmers control to deactivate this default to get a smaller memory footprint. It is also simple to add a read-only block of memory to the processor system. If the **.text** and **.rodata** segments are assigned to this block in *BSP Editor*, the load segments for global data will also be placed in this ROM, protecting a fully restored program after reset.

TABLE II
ALTERA EMBEDDED TOOLCHAIN

Core	Nios II/e 13.1
HW tools	Quartus II / Qsys 13.1 64-bit
SW tools	Nios II 13.1 SW Build Tools for Eclipse BSP Editor / BSP Generator
Core	Nios V/m 2.0.0
HW tools	Quartus Prime / Platform Designer 23.1.std
SW tools	Ashling RiscFree IDE for Intel FPGAs BSP Editor / BSP Generator
Board	Terasic Altera DE2-115
FPGA	Cyclone IV-E EP4CE115F29C7

B. Microchip (former MicroSemi, Actel)

Microchip offers the *Mi-V RV32* RISC-V derivative. It can be equipped with ROM and RAM blocks. However, Microchip is the only one of the surveyed vendors that does not generate a tailored BSP for the designed system. Software parameters like peripheral addresses have to be edited by hand. The default linker script for software projects does not include loadable sections, but another version with split ROM/RAM and load segments is available as a template. The default startup code reinitializes global data if it finds a load segment.

TABLE III
MICROCHIP EMBEDDED TOOLCHAIN

Core	Mi-V RV32I 3.1
HW tools	Libero SoC / SmartDesign v2023.2
SW tools	SoftConsole v2022.2-RISC-V-747

C. Lattice

As an successor to their legacy *LatticeMico32* processor, Lattice has also introduced their own RISC-V design with three different resource/performance variations (SM/MC/RX). Default generated linker scripts have no load sections and the provided startup code only zeroes the **.bss** segment. The *Propel* system design tool can add read-only memory blocks, but there is no automatic assignment of constant data or code to those blocks.

TABLE IV
LATTICE EMBEDDED TOOLCHAIN

Core	RISC-V SM/MC/RX
HW tools	Diamond 3.13 / Lattice Propel 2024.1
SW tools	Lattice Propel SDK 2024.1

D. Efinix

Efinix' *Sapphire SoC* soft processor is based on the open source *VexRiscv* [9] core. The embedded flow generates a BSP with a linker script with no loadable sections, however the default boot code supports initialization of globals if a load segment is found. No ROMs can be specified in the IP generator for the *Sapphire* system.

TABLE V
EFINIX EMBEDDED TOOLCHAIN

Core	Sapphire RISC-V SoC v3.0.0
HW tools	Efinity 2023.2.307
SW tools	Efinity RISC-V Embedded Software IDE 2023.2.1.1

E. Overview

Table VI-E gives a systematic overview about which soft processor toolchains support which features to ensure correct runtime behaviour, even after a soft reset. We have also added a rubric indicating that we only had access to AMD and Altera hardware to test runtime behaviour in practice. The information for the other three vendors is based on detailed analysis of the toolchains and generated code.

VII. CONCLUSION

We have illustrated that a simple soft reset can cause problematic software behaviour in an embedded application if the useful properties of FPGA block RAM are used carelessly. This is at least partially true for four of the five FPGA vendors we surveyed. It is unclear if these are deliberate design choices with a focus on memory footprint or oversights.

We strongly recommend that AMD emulates Altera in offering load segments and boot code initialization of the data segments - preferably as an option - during BSP generation. There is also no compelling reason why the current hardware design process in IP Integrator makes it infeasible to include a true ROM component in the Local Memory Bus block. We hope that AMD fixes both issues not just for the legacy Microblaze but also for the new RISC-V core.

While we have not looked in as much detail at Microchip, Lattice and Efinix, mostly due to a lack of hardware, we see no reason not to add the loadable segment option for their linker script generation, either. It is also in Microchip's own interest to catch up to its competitors in terms of Board Support Package generation.

ACKNOWLEDGMENTS

This project has been conducted without any direct or indirect industry funding. However, we would like to thank all mentioned FPGA vendors for providing their tools and IP for free (in some cases for a limited evaluation period).

	Altera Nios II/Nios V	**AMD/Xilinx** Microblaze/MB-V	**Efinix** Sapphire	**Lattice** RISC-V SM/MC/RX	**Microchip** Mi-V RV32
Behaviour examined in HW / only in SW tools	HW	HW	SW	SW	SW
Board Support Package tailored to HW	✓	✓	✓	✓	✗
Optional load segments in linker scripts	✓	✗	✗	✗	example
Boot code can re-load *.data* segment	✓	✗	✓	✗	✓
Read-only boot memory supported in HW design	✓	✗	✗	✓	✓
Load segments auto-assigned to read-only memory	✓	-	-	✗	example

References

[1] ISO/IEC 9899:2018(E), "Information technology - Programming languages - C", International Standards Organisation, Geneva, Switzerland, 2018

[2] R. M. Stallman, "Using the gnu compiler collection.", Free Software Foundation 4, no. 02, 2003.

[3] TIS Committee, "Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification, Version 1.2", 1995, https://refspecs.linuxfoundation.org/elf/elf.pdf (accessed Mar. 29, 2024).

[4] "The Newlib Homepage", https://sourceware.org/newlib/ (accessed Mar. 29, 2024).

[5] "crtinit.S", Xilinx, 2001/2009, https://github.com/Xilinx/newlib/blob/xlnx/newlib-2_1-branch/libgloss/microblaze/crtinit.S (accessed Mar. 29, 2024).

[6] "7 Series FPGAs Memory Resources User Guide (UG473)", Xilinx, 2019

[7] "Github repository with Microblaze fixes", https://github.com/rw-hsma-fpga/microblaze-globalvar-fixes (accessed Aug. 3, 2024).

[8] A. Waterman, Y. Lee, D. Patterson, K. Asanovic, "The RISC-V instruction set manual. Volume I: User-Level ISA", version 2.0, 2014, pp.1-79.

[9] "SpinalHDL VexRiscV", https://github.com/SpinalHDL/VexRiscv (accessed Mar. 29, 2024).