Relocatable RISC-V Rust Applications for Embedded Systems

Hugo McNally <hugom@lowrisc.org>
Luís Marques <luismarques@lowrisc.org>

Problem

The RISC-V ecosystem currently lacks an Application Binary Interface (ABI) that adequately supports common requirements for relocatable applications in embedded systems.

Requirements:

- No Memory Management Unit (MMU)
- Execute In Place (XIP)
- RAM and ROM segments can be relocated independently

The existing RISC-V System V ABI does not allow satisfying all of these at the same time, as it assumes the use of a single relocation offset (base address) for the entire application.

Examples

The following assembly examples are implementations of this C function, where x is a global variable.

```
int get_x() {
    return x;
}
```

RISC-V Absolute Addressing

If the address of **x** is independent of application position, e.g. it's a memory-mapped device, one can continue to use absolute addressing.

```
lui  a0, %hi(x)
lw   a0, %lo(x)(a0)
ret
```

RISC-V PC-Relative Addressing

If **x** is within 2 GiB of the PC, PC-relative addressing can be used. This is because the offset from the PC is independent of the application position.

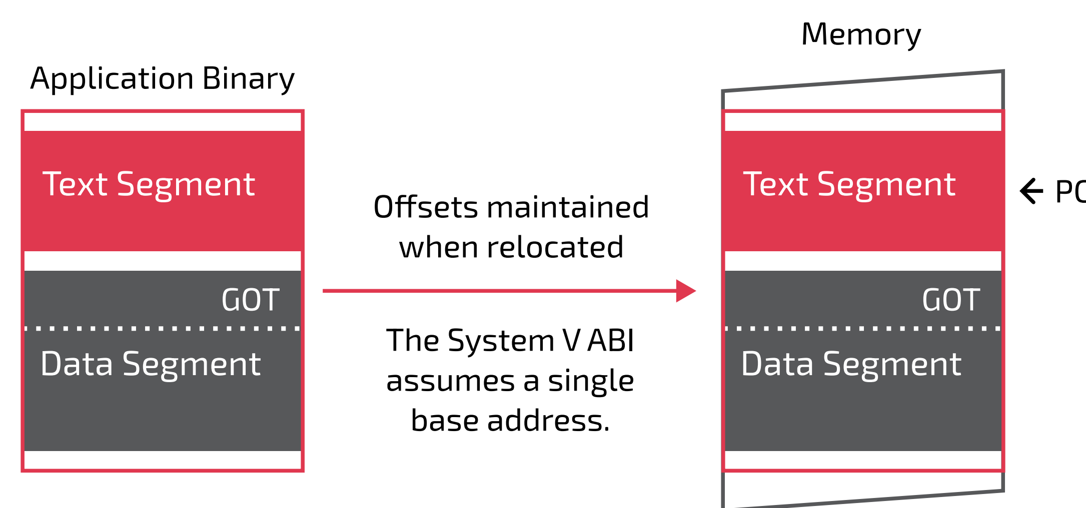
```
1: auipc a0, %pcrel_hi(x)
   lw    a0, %pcrel_lo(1b)(a0)
   ret
```

RISC-V System V ABI

The System V ABI enables creating relocatable applications. For RISC-V, this is typically done by using PC-relative addressing when x is within 2 GiB of the PC and by using a Global Offset Table for when **x** is not local (preemptible).

The GOT is filled with the absolute address of each global (preemptible) symbol by the application loader. It's position is known relative to the PC.

```
1: auipc a0, %got_pcrel_hi(x)
   lw a0, %pcrel_lo(1b)(a0)
   lw a0, 0(a0)
   ret
```



Why can't the System V ABI be used?

Although using the GOT solves the problem of having independently relocatable text and data segments, the ABI relies on the GOT itself being at a known offset relative to the PC, which means the text and data segments cannot be independently relocated. FDPIC Style ABIs solve this.

RISC-V GP-Relative Addressing

GP-relative addressing can be used when **x** is within 2 GiB of the global pointer (GP).

```
lui  a0, %gprel_hi(x)
add  a0, gp, a0
lw   a0, %gprel_lo(x)(a0)
ret
```

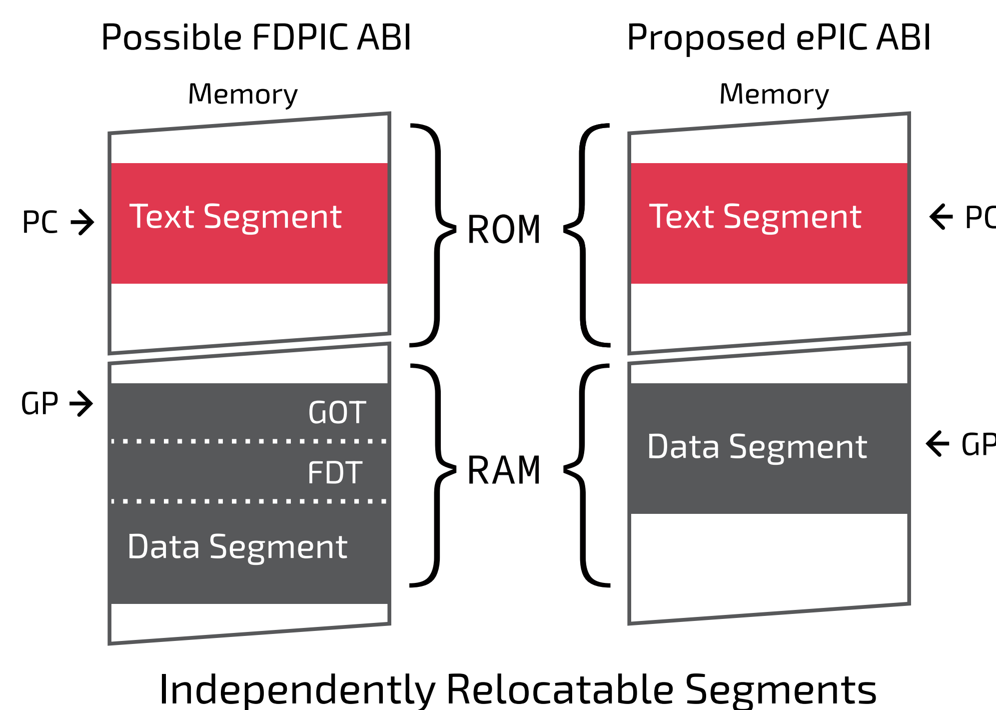
Possible RISC-V FDPIC ABI

FDPIC allows the text and data segments to be independently relocatable, by reserving a register, the global pointer (GP), to point to a known position in the data segment. This means the GOT can be found relative to the GP and not the PC.

With this ABI, *PC-relative addressing* or *GP-relative addressing* can be used when **x** is a local symbol. It must be within 2 GiB of the PC or GP.

When x is external (preemptible), the address can be computed using the GOT.

```
lui  a0, %gprel_got_hi(x)
add  a0, gp, a0
lw   a0, %gprel_got_lo(x)(a0)
lw   a0, 0(a0)
ret
```



Independently Relocatable Segments

The Proposed ePIC ABI

The proposed ePIC ABI uses a combination of PC- and GP-relative addressing, similar to FDPIC local addressing. Since we do not require shared library support, we can avoid paying for the cost of the GOT, used for external (preemptible) data addressing.

Sometimes only the linker knows whether a symbol will reside in the text segment or data segment. For these situations ePIC has instruction-rewriting relocations, allowing the linker to select between PC-Relative or GP-relative addressing depending on the symbol's segment.

```
1: lui  a0, %epic_hi(x)
   add  a0, gp, a0, %epic_base_add(x)
   lw   a0, %epic_lo(1b)(a0)
   ret
```

```
1:
auipc a0, %pcrel_hi(x)
nop    // can be removed
lw     a0, %pcrel_lo(1b)(a0)
ret
```

```
lui  a0, %gprel_hi(x)
add  a0, gp, a0
lw   a0, %gprel_lo(x)(a0)
ret
```

Why ePIC is being proposed over potential alternatives such as an FDPIC-style ABI

Memory Usage: FDPIC uses a GOT which has an entry for each global symbol. This costs one word per entry (RV32).

Start-up Cost: FDPIC requires the GOT to be filled by the loader at load time, increasing the startup time.

Simplicity: The full generality and cost of FDPIC is not needed, as support for shared libraries is not required.

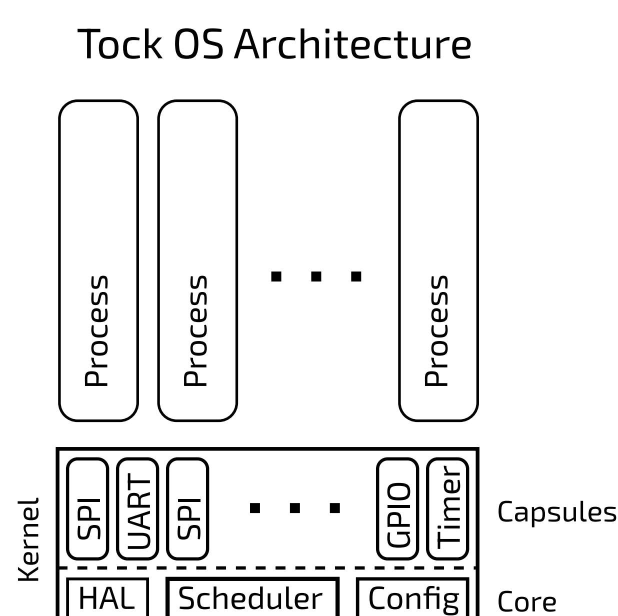
Access Cost: PC-relative and GP-relative accesses are cheaper than GOT based accesses. Compare the instruction sequences' lengths, noting the additional load instruction.

The Security Tock OS

The Rust Language enforces more restrictions than are common in systems level languages, but these restrictions provide guarantees of correctness from the compiler. These correctness guarantees greatly reduce security risks.

Tock OS is an Operating System built to capitalise on these features. It only opts out of these restrictions (using the *unsafe dialect*) where it is necessary in the core of the kernel. Most functionality is managed in Capsules (Drivers) which have the restrictions enforced. Processes are dynamically loaded by the kernel. They are isolated using the MPU and scheduled preemptively to have stronger system liveness guarantees.

Tock OS Architecture



The Motivation behind ePIC

Processes being dynamically loadable allows them to be built, signed and updated separately from the kernel.

Tock OS allows multiple applications to be loaded at the same time. Because there is no MMU, to make sure applications don't collide with one another, they have to be relocatable.

ePIC enables efficient relocatable applications for Tock OS that can be written in Rust to take full advantage of its correctness guarantees.

The ePIC Spec