RISC-V Summit North America 2022

This reports includes summaries of three selected talks and a hands-on session from the RISC-V Summit North America 2022:

- **RISC-V and the Elf: A Story about Randomisation for Safe and Secure Critical Systems**: the more technical of the three talks, it focuses on the practice of software randomisation, what it is and how it is relevant vis-a-vis secure critical systems; the talk revolves around a publication on the matter from the talk's speaker, Leonidas Kosmidis from BSC
- **Is RISC-V HPC? RISC-V is HPC!**: an introspective look into open-source hardware, HPC and related European initiatives from the viewpoint of John Davis, SIG HPC Chair and also from BSC
- **RISC-V for Aerospace and Defense Applications**: an very interesting talk from SiFive executive Tom Leahy on different funding opportunities for RISC-V initiatives under the umbrella of the US's CHIPS Act 2022 and how the Aerospace and Defense (A&D) market (at commercial and governmental levels) is specially interested in the vertical-integration possibilities that an open ISA provides, not only in meeting highly-specific requirements from demanding and solemn defense-related projects, but also in the industrial and commercial autonomy a national semiconductor industry paired with a "domestic" ISA could bring to the table (a bit number-heavy, but the between-the-lines context and tone of the talk was very telling and interesting in my view)
- **Toolchains tutorial**: a hands-on session working with the riscv-gnu-toolchain project, as well as an introductory part on toolchain-related concepts, terminology and definitions

1. RISC-V and the Elf: A Story about Randomisation for Safe and Secure Critical Systems

Speaker: Leonidas Kosmidis - Barcelona Supercomputer Center (BSC) & Universitat Politécnica de Catalunya (UPC)

1.1. Introduction

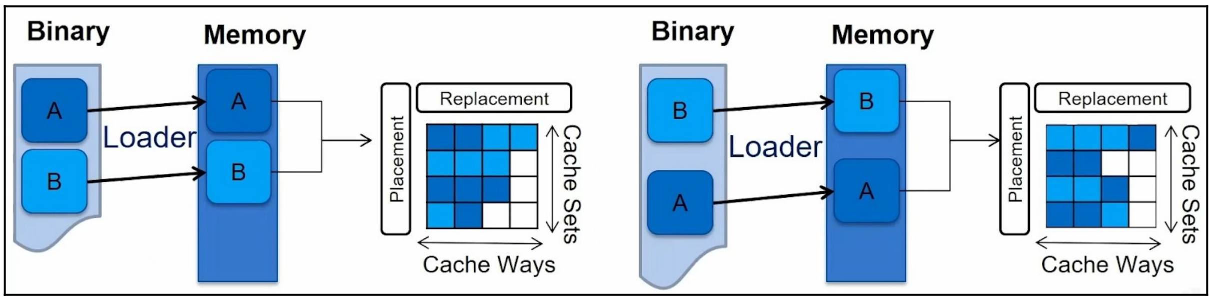
Software randomisation consists of the practice of randomly varying a program's characteristics without altering its core functionality or its final outcome. A program's memory layout is an example of what can be considered for this endeavour. This is relevant in some fields and/or to certain ends (some references were included in the talk to support such claims):

- Worst-case execution time (WCET) and probabilistic timing analysis of workloads in real-time systems, see "TASA: Toolchain-Agnostic Static Software Randomisation for Critical Real-Time Systems", ICCAD 2016
- Performance evaluation of platforms while accounting for platform jitter, see "STABILISER: Statistically Sound Performance Evaluation", ASPLOS 2013
- System security enhancement to mitigate exploits based on knowledge of timing and/or program layout, see "KASLR Is Dead: Long Live KASLR", ESSoS 2017

1.2. Software randomisation

The talk focuses at first on defining static software randomisation. The basic idea behind it involves creating binary files with a different, random binary layout from the original one, all before actual program execution. In this context, randomising a binary file's layout is synonymous with randomising its memory layout.

As the program is loaded into memory, changing a program in a functionally-consistent fashion will have different sections of the program mapped to different, specific sections in memory. Different sections of a program are "accessed" more/less frequently than others, which implies different cache-hit/miss rates, and, as different program sections are mapped in different sections of memory, this results in different program sections conflicting or interacting with each other in varying ways. Different mappings of program sections in memory result in differing amounts of cache-hits/misses for a program, with their consequent variations in execution times.



The talk then visits the concepts behind one of the papers referenced in the previously, the TASA paper presented at ICCAD 20126. After static software randomisation comes a variation of it in the form of static software randomisation at source code level. If static software randomisation operates on binary files, this variant deals in compiler-level mechanisms. This is the concept introduced in the TASA paper. If compilers generate program elements in the order they are encountered within a source file, then modifying the order within the source file itself will alter how a compiler organises a binary file.

1.3. Randomisation and the Elf binary file format

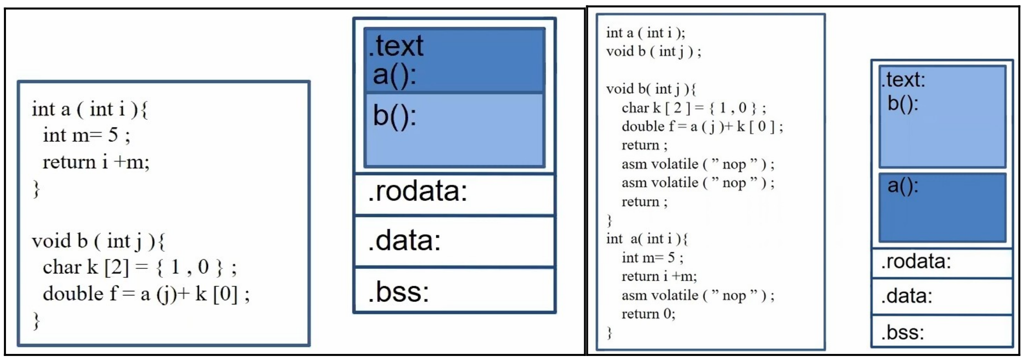
The Elf binary file format (“.elf” files) is one of the most used binary file formats to date, from bare metal embedded systems up to full Linux-capable systems. An Elf file is divided into four main sections:

- “.text” section: for code
- “.rodata” section: read-only global data such as strings or constants
- “.data” section: for initialised, global variables whose value differs from zero
- “.bss” section: for uninitialised, global variables or initialised, global variables whose value has been set to zero

An Elf file’s sections are loaded in memory before program start-up, such that each entry of an Elf file’s sections must be aligned to its corresponding size. The “.text” and “.rodata” sections are placed together as operating systems treat them both as read-only data, simplifying its due protection process.

Building upon static software randomisation at source code level, a way of modifying an Elf binary file’s memory layout can consist simple code modifications such as:

- Reordering functions at random
    - Function prototypes are needed at the beginning of the source code file to allow for this
    - This technique shuffles the mapping of source code within the binary file and, consequently, its placement in memory
- Introducing randomly-sized padding within function definitions via NOP instructions
    - This effectively changes the size of function definitions while keeping function behaviour unchanged
    - NOP inline assembly instructions should be added after “return” statements so that they are never executed
    - Randomly varying the size of functions has a cascading effect on the placement of the rest of Elf-file sections, as different function sizes shift the remaining sections “up” or “down” in memory

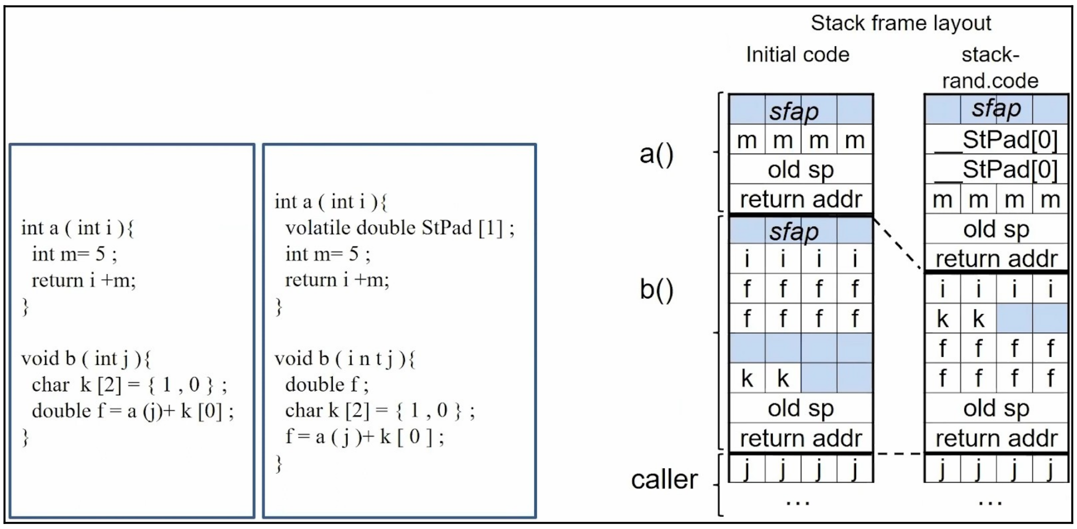


```
int a ( int i ){
  int m= 5 ;
  return i +m;
}

void b ( int j ){
  char k [2] = { 1 , 0 } ;
  double f = a (j)+ k [0] ;
}
```

```
int a ( int i );
void b ( int j ) ;

void b( int j ){
   char k [ 2 ] = { 1 , 0 } ;
   double f = a ( j )+ k [ 0 ] ;
   return ;
   asm volatile ( ” nop ” ) ;
   asm volatile ( ” nop ” ) ;
   return ;
}
int  a( int i ){
   int m= 5 ;
   return i +m;
   asm volatile ( ” nop ” ) ;
   return 0;
}
```

Original source code (left) and reordered+padded functions (right)

Randomisation of the stack frame must also be considered. Local variables and their values reside within the stack frame, and the size of each stack frame depends on the size of its local variables, accounting for size-dependent variable aligning and also the alignment of the stack frame as well (Stack Frame Alignment Padding, sfap). TASA uses two complementary solutions regarding the management of the stack frame: inter-stack and intra-stack randomisation. Inter-stack randomisation increases the size of a stack frame by an arbitrary amount, which, in much the same way as with function blocks, implies a variation in the resulting conflicts between different stack frames, resulting in execution time changes. Intra-stack randomisation, on the other hand, deals in the reordering of local variables’ declarations, again, similarly to function definitions in source code files. Reordering the declaration of local variables, as with the other previous reordering efforts, causes a variation of conflicts between local variables and, as a result, timing changes in the execution of the program. The size of the stack frame can also vary with intra-stack randomisation due to the compiler trying to align variables in memory according to their size. The figure below illustrates this by adding inter-stack randomisation in the form of the introduction of an unused array in the definition of function a() and by adding intra-stack randomisation in function b() by changing the ordering of local variables. The randomisation of the stack frame also affects the rest of the Elf file sections in a cascading manner with each size variation.



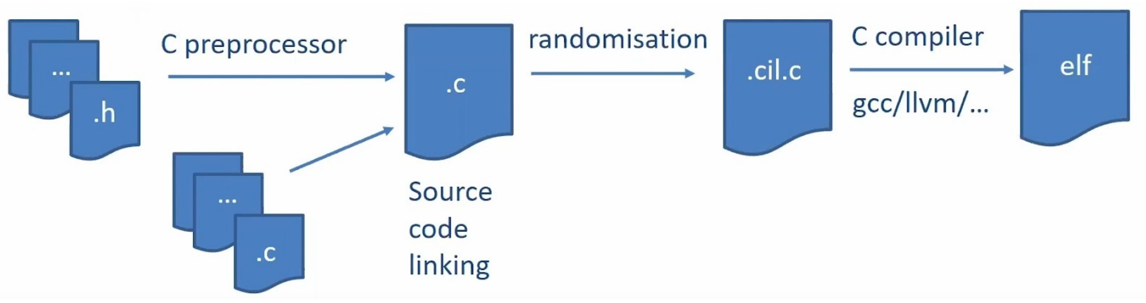
```
int a ( int i ){
  int m= 5 ;
  return i +m;
}

void b ( int j ){
  char  k [2] = { 1 , 0 } ;
  double f = a (j)+ k [0] ;
}
```

```
int a ( int i ){
  volatile double StPad [1] ;
  int m= 5 ;
  return i +m;
}

void b ( i n t j ){
  double f ;
  char k [2] = { 1 , 0 } ;
  f = a ( j )+ k [ 0 ] ;
}
```

Original and randomised code (left) and stack frame view of each (right)

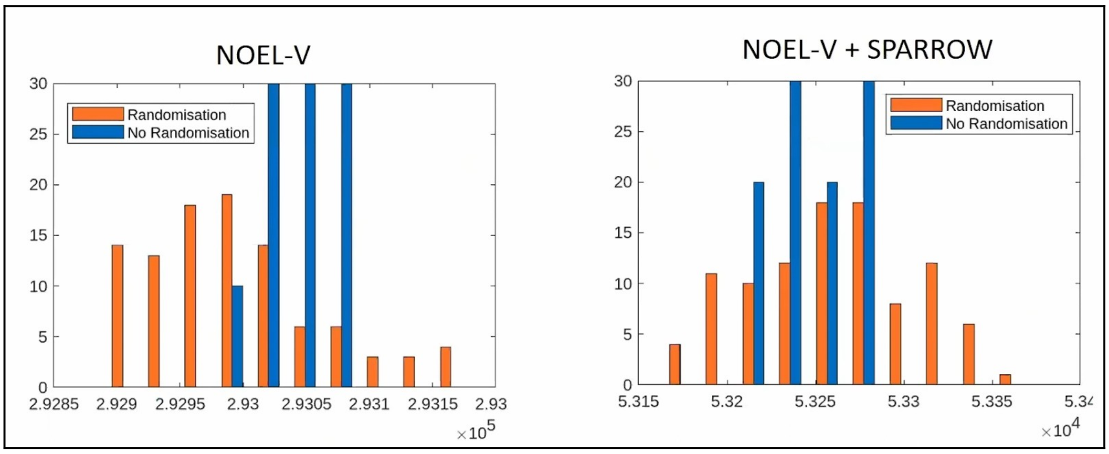
Global variables and program data randomisation follow the same concept and can also be shuffled, although with slight differences. Global variables are placed in different sections of an Elf binary file depending on whether they are read-only, whether they are initialised and whether the initialisation value corresponds to zero or not. TASA distributes the global variables into the “.rodata”, “.data” or the “.bss” sections depending on the variable’s specifics and it reorders the variables within the segment, taking advantage of the compiler-induced alignment of data.

TASA is implemented within the CIL source-to-source C compiler such that it sits between the C preprocessor and linking steps and the actual compilation of the “.c” file into an Elf file. TASA supports any C construct, including gcc extensions and is open source.



1.4. Randomisation demonstration on NOEL-V

In order to showcase the results of this randomisation process, the NOEL-V processor was implemented in a Xilinx Ultrascale+ FPGA in its TIN32, Single-Issue configuration at 100MHz with 2-way instruction and data caches. Within this processor was also added the SPARROW vector AI accelerator to demonstrate the compatibility of software randomisation with the extension of processors with additional ASICs. The standalone processor and its extended version were tested both with and without randomisation to show the different variability in timing when executing the same workload on a randomised versus a non-randomised version of the experimental setup. The randomised version of both the NOEL-V and the extended NOEL-V have flatter histograms for execution times of the tested workload, while the non-randomised are much more stable in their yielding of execution times.

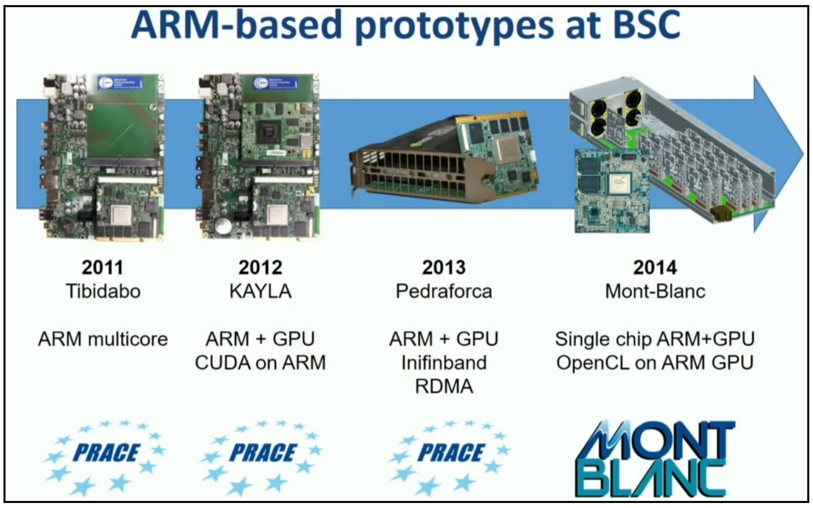


2. Is RISC-V HPC? RISC-V is HPC!

Speaker: John Davis - Barcelona Supercomputer Center (BSC) & RISC-V International SIG-HPC Chair

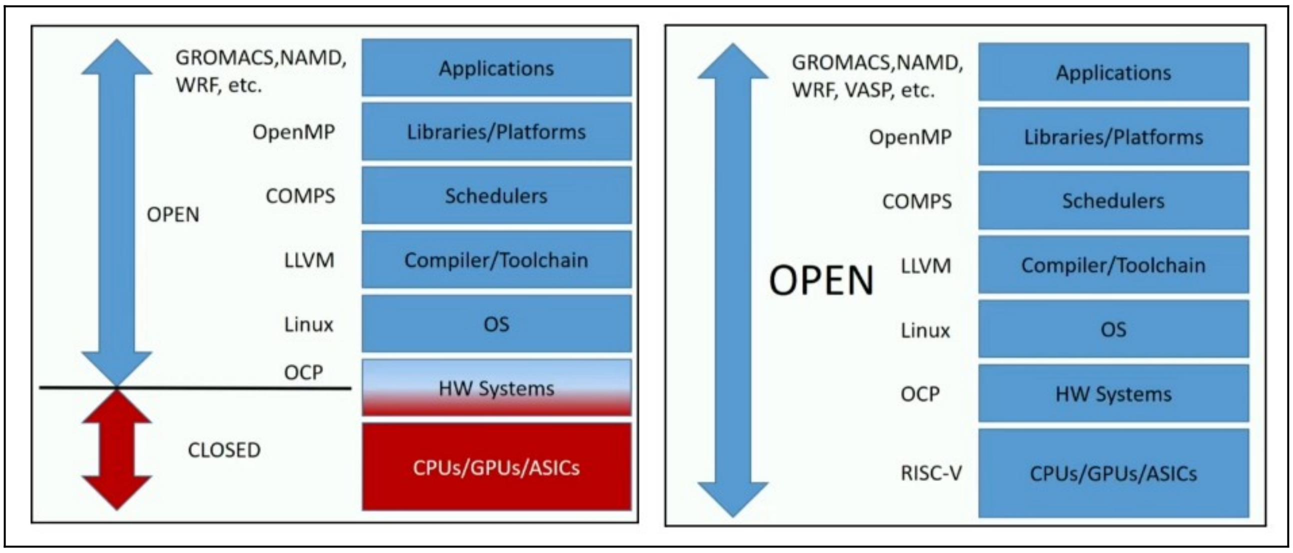
2.1. The Open HPC Stack Vision

Back in the 2008-2010 range, BSC was entrenched in promoting ARM as the basis for an HPC system. From these efforts, a series of prototype systems emerged based on ARM architectures so as to develop the software stack for HPC in the ARM ecosystem. These projects include the Tibidabo project (2011), KAYLA (2012), Pedraforca (2013) and Mont-Blanc (2014).



The final project of this lineage, the Mont-Blanc project, was the most relevant. It ended in 2020, after some years of intense collaboration between European partners dedicated on specific sections of the software stack revolving around the Mont-Blanc initiative. BSC was tasked with meddling in the system software layer and the hardware layer. However, useful though the project proved to be, the lack of configurability and tinkerability of a fixed ISA constrained some research and overhaul avenues discerned throughout the course of the project. The initial focus of the ARM lineage of projects was then followed by the Fugaku supercomputer in Japan, which managed to secure the first spot on the TOP500 list for several years in a row.

From the restrictive nature surged the need and desire of an open-source hardware-software ecosystem that could enable free movement from a designer's perspective across the entirety of the stack: from the CPU itself up to the applications running on them and everything in between. To this end, Europe finds itself in the position of being a heavily incentivised agent to promote the definition of a common, home-grown open HPC hardware-software ecosystem. Along the lines of this initiative's philosophy, understandably, Linux stands as the de facto standard OS, despite propietary alternatives. Providing common platforms, specifications and interfaces between layers and components is key for the long-term survivability of the effort, as well as for building new functionality by leveraging existing components and long-standing projects/ecosystems. The openness at the core of the oportunity this initiative provides is applied at the smaller and larger scales, from RTL code all the way up to the highest-level of programming languages. Hardware can benefit from the long-standing tradition of open-source interoperability and contribution efforts in much the same way that the giants of open-source have until today.



RISC-V provides the open source standard for a hardware alternative to dominant, proprietary, non-EU solutions. With the fostering of a RISC-V ecosystem, Europe can achieve technological sufficiency from (or at least substitute) the currently standing foundational building blocks of the HPC ecosystem (among others). Comparatively speaking, RISC-V now stands before many of the same adoption hurdles in which Linux was many years ago. RISC-V can unify, focus and build a new microelectronics industry in Europe.

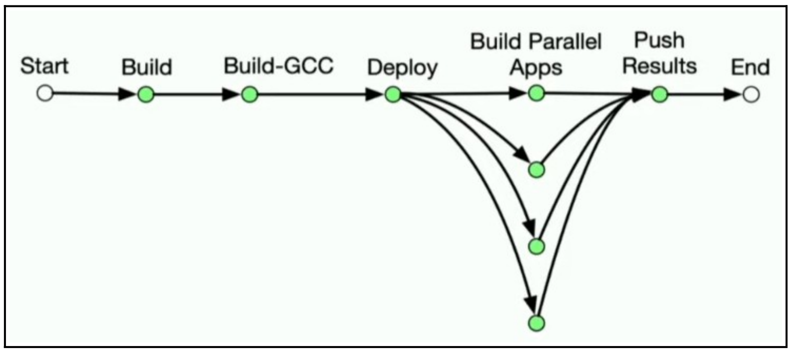
2.2. SIG HPC

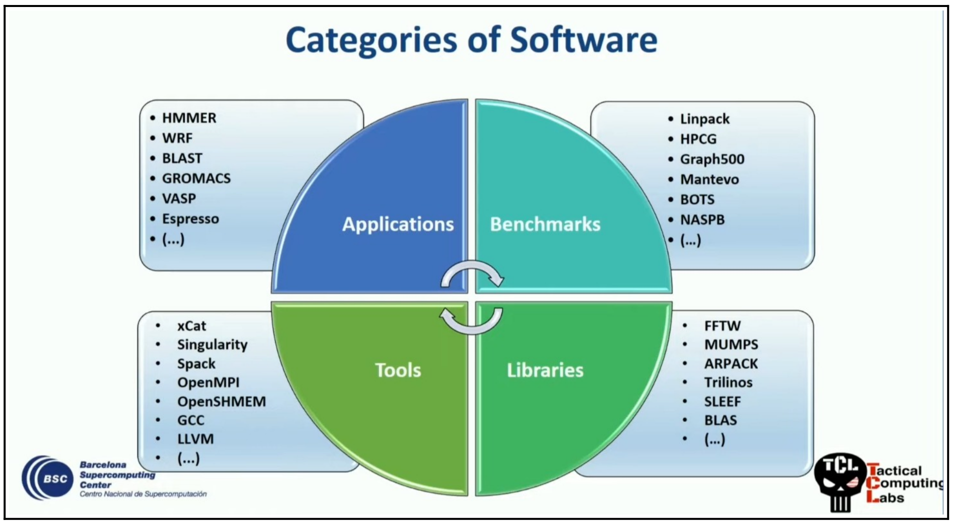
Within the RISC-V International Consortium, there is a specialized group focused on the development and research of RISC-V on the HPC track: the Special Interest Group (SIG) for HPC, of which the speaker is a current Chair. The SIG-HPC is fixed on technical and strategic imperatives that guide the RISC-V ecosystem development to enable an open HPC ecosystem. The mission of the group is to enable and explore the open source ISA in a broader set of new software and hardware opportunities in the HPC solution space, from the edge to supercomputers, to allow the software ecosystem atop the lower hardware levels to flourish under a more accessible and open standard that supports both legacy and emerging HPC workloads.

This implies that the Special Interest Group within the RISC-V Consortium fixates on supporting the development and tracking the progress of CPUs, accelerators, specific IPs and coprocessors, as well as verification and compliance methodologies pertinent to HPC systems. The group also oversees hardware and software alignment, keeping track of software ecosystem support and IP incorporation into the RISC-V ecosystem whenever HPC may benefit from it. It also undertakes engagement and representation efforts in compute-intensive fields or industry and academic events by bolstering and championing the influence of RISC-V, while always on the lookout for potential industrial and academic partners aligned with the group’s mission.

Some of the initiatives of interest for the group have been the virtual memory extension proposals for RISC-V in the SV57, SV57K, SV64 and SV128 versions, developing a set ecosystem guidelines and roadmaps to guide and empower the community in the development of a thriving SW and HW ecosystem, the group has also overlooked the development of accelerators and ISA extensions and has started with the enabling of porting HPC libraries and RISC-V testbeds to lay the groundworks for the HPC software stack. Finally, the group also concerns itself with the development of adapters and more generic devices such as interconnection networks, GPUs, etc.

There exists a Jenkins-based continous integration pipeline built by Tactical Computing Labs in collaboration with the SIG-HPC group that tests whether tools and projects compile and work successfully for RISC-V architectures. This is an example of what the HPC software stack testbed should strive to support in order to drive requirements for different working groups in both academia and industry. By provinding an HPC-centric, RISC-V-based software test suite supporting different compilers (gcc, llvm, etc.), target libraries and whole projects can be cross-compiled for benchmarking and compatibility purposes, while being able to specify the target RISC-V. The first objective of stablishing the HPC testbed, however, will revolve mainly around guaranteeing support for the major low-level libraries that HPC workloads typically use (GROMACS, BLAS, OpenMPI, BLAST, etc.).





- HMMER
- WRF
- BLAST
- GROMACS
- VASP
- Espresso
- (...)

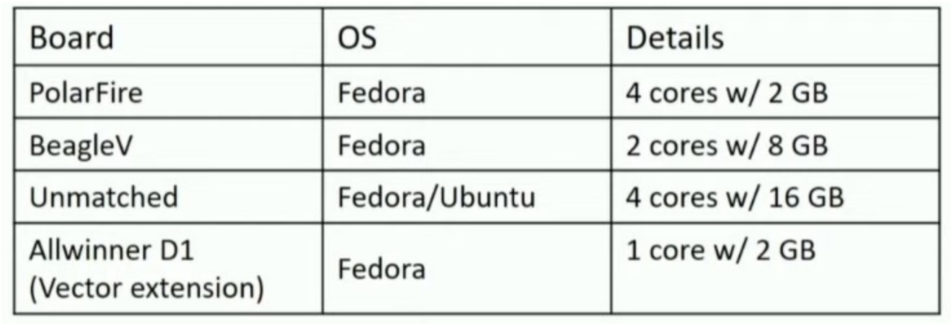
- Linpack
- HPCG
- Graph500
- Mantevo
- BOTS
- NASPB
- (...)

- xCat
- Singularity
- Spack
- OpenMPI
- OpenSHMEM
- GCC
- LLVM
- (...)

- FFTW
- MUMPS
- ARPACK
- Trilinos
- SLEEF
- BLAS
- (...)

The project itself, riscv-hpc, is available through Github for internal, propietary use as well as for public contributions to the project. The harness includes a set of template scripts for each test, such that each test pulls its own source code, it unpacks and builds it and test scripts are added as simply as including them in a pull request for the Jenkins pipeline.

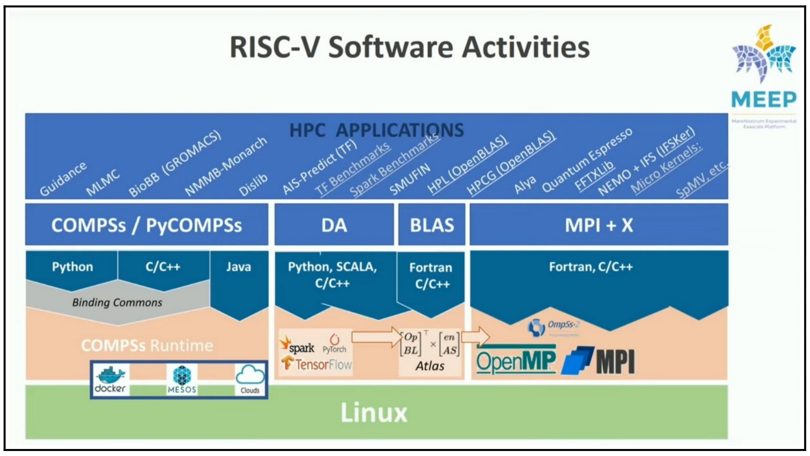
On the hardware side, there are two specific hardware testbeds in use in Europe to-date: SUPER-V from BSC and the ExCALIBUR H&ES RISC-V Testbed based in the UK. Soft-cores and physical boards tested by the SIG-HPC group include the PolarFire, BeagleV, Unmatched and Allwinner D1 boards, all running Fedora/Ubuntu operating systems with a varying number of cores.



Board	OS	Details
PolarFire	Fedora	4 cores w/ 2 GB
BeagleV	Fedora	2 cores w/ 8 GB
Unmatched	Fedora/Ubuntu	4 cores w/ 16 GB
Allwinner D1 (Vector extension)	Fedora	1 core w/ 2 GB

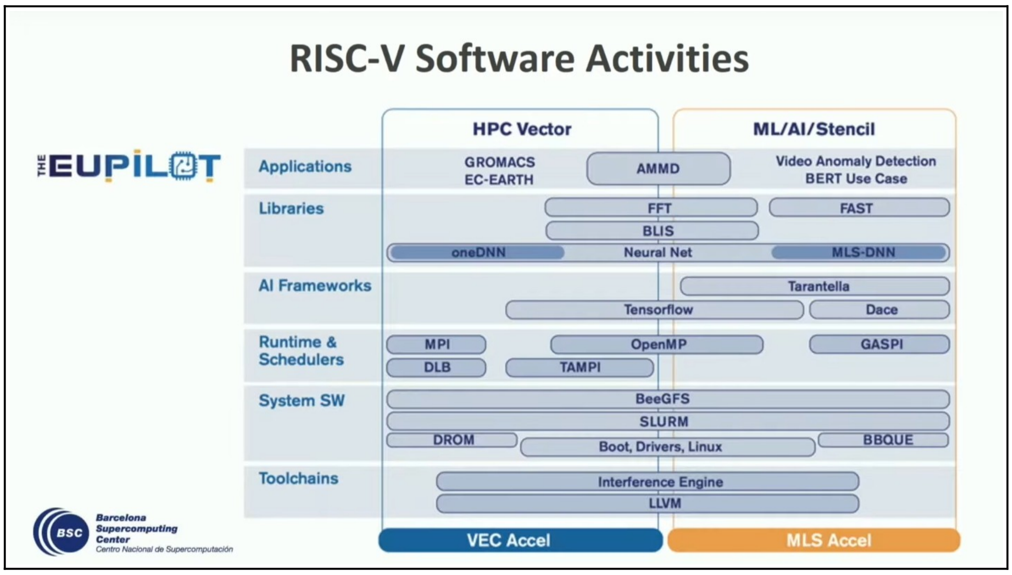
The European Processor Initiative (EPI) has also enabled and bolstered the adoption of many libraries for the above platforms through funding and collaboration efforts in the shape of available FPGAs running soft-cores or QEMU-based simulators for testing.

2.3. RISC-V research: RISC-V is HPC!



It wasn't until 2021 that the Open Source Hardware (OSH) initiative stuck on and various roadmap reports started to surface concerning European interests and RISC-V-based hardware and software development. In 2022, the OSH philosophy and terminology starts to appear in the KDT JU Work Programme and the EU Chip Act as an avenue of action from both research and industrial perspectives. These reports include the EPI initiative/project, as well as other efforts such as the MEEP project (Marenostrum Experimental Exascale Platform), to do with using 100+ FPGAs to emulate RISC-V accelerators and drive a larger testing and execution ecosystem; eProcessor, to do with building open-source, out-of-order cores; EUPILOT, derivative of the EPI project focused on larger-scale systems and the ISOLDE project from KDT, focused on high-performant, secure RISC-V processors. The EPI project now ships a four-core, variable-precision, vector processor with ML and Stencil acceleration capabilities, the MEEP project develops vector and systolic-array coprocessors, the eProcessor issues OOO single and multicore chops and EUPILOT deals in accelerator systems. On the software side, the MEEP and EUPILOT have made progress on supporting HPC-relevant libraries and frameworks. Other funding opportunities from the EU are in the works, as Europe has demonstrated an inclination for the initiative.

Apart from European projects themselves, there are a bunch of companies associated with the RISC-V Consortium as strategic members fully devoted to the development of application-class, out-of-order, vector-capable machines, which signifies a thriving and vibrant common enterprise is in the works in the RISC-V initiative as a whole.



RISC-V Software Activities

3. RISC-V for Aerospace and Defense Applications

Speaker: Tom Leahy – SiFive's Head of Aerospace and Defense

3.1. High-level aerospace and defense (A&D) challenges

As opposed to commercial pursuits, where time-to-market is key and iterations/generations of IPs go by in a matter of months in an endless race to overtake competitors' potential advancements, the aerospace and defense (A&D) sector has traditionally refrained from operating under such deadlines. Recently, however, the A&D sector has started looking for a decrease in program and release schedules in favour of veering towards commercial timescales, prancing along the blosoming of RISC-V and investment opportunities around it. The A&D sector is looking to engange in providing an asymmetrical technical advantage for any and all future projects such that the computing aspect of any of them can be readily maintained to keep up with the state of the art while, at the same time, not needing to relinquish the vertical-integration-provided platform control that an open-source ISA enables in contrast with commercial-off-the-shelf products.

The sector in question is also in pursuit of an extension of project longevity (in the 15-year-and-more range), demanding flexibility, scalability, upgradeability and re-use of all aspects of the hardware-software stack. The A&D sector must also deal with cross-functional teams across different governmental departments such as the Department of Defense (DoD), Department of Commerce (DoC) and the Department of Energy (DoE), without compromising, if possible, in the ways of size, weight, power and cost requirements (SWaPC requirements). All of this is sought to be accomplished with a tight partnership between the public and private sectors.

More specifically, low-level requirements include, but may not be limited to:

- Lower PPA (Power, Performance, Area) specs for projects to address SwaPC concerns
- Openness of standards and overall ecosystem to foster development, growth and talent acquisition in a manageable IP ecosystem
- Improving security of solutions
- Improving AI/ML processing
- Focus on radiation-hardened solutions
- Growth of the chiplet approach and modular packaging infrastructures
- Engangement in the CHIPS Act, National Semiconductor Technology Centres (NSTCs) and Microelectronic Commons for national (US) microelectronics production

3.2. Investments being made at national (US) level

Investments being made in the fields of semiconductor manufacturing or IP development are being handled as a matter of national security. As such, it is treated with the due severity they consider it deserves: the CHIPS Act funding reaches the $54.2B dollar mark, with the European Chips Act at a close €43B, Chips 4 Alliance encompasses the collaboration and relations between the USA, Japan, South Korea and Taiwan… This can be framed as a more-than-bountiful opportunity for the semiconductor and design industries, brought about by the sparsity and decentralization of supply chains as a result of neoliberal economic policies applied to what now is perceived as instrumental for the safeguarding of interests of the USA.

As a direct consequence of this, where in the 1990s the USA was responsible for 30% of the global semiconductor supply and now it stands at the 11%, numerous fab-related expansions are being announced and offered to key players in the semiconductor and packaging industries (via, presumably, tax exemptions and subsidies) to lay the groundworks for a USA-based industrial backbone of semiconductor manufacturing and packaging:

- $20B for Ohio-based Intel facilities
- $30B for Arizona-based Intel fab expansion
- $100B for New York-based Micron facilities
- $15B for New York-based GlobalFoundries facilities
- $191B for Texas-based Samsung facilities
- $30B for Texas-based Texas Instruments facilities
- $1.2B for New Jersey-based Skyworks facilities
- New Mexico-based Intel-led advanced packaging operation facilities
- Florida-based SkyWater-led advanced packaging operation facilities
- Amkor in talks for opening first USA-based plant
- Northrop Grumman in talks for opening a packaging processing line
- …

The funding of the CHIPS Act is set at $54.2B, which is just a subset of the overall funding of the (more general) CHIPS & Science Act of 2022, which is structured as follows:

- Division A: total budget of $78.2B
  - CHIPS Act 2022: $54.2B
  - Investment Tax Credit (ITC): estimated $24B
- Division B – Research & Innovation: $169.9B
- Division C – Supreme Court Supplemental Appropriations: $20M
- Total budget: $248.12B

3.3. US-based microelectronics funding areas

The totality $54.2B allotted to the CHIPS Act 2022 initiative is divided among different parties as follows (the underlined-and-bolded points are where RISC-V initiatives can obtain funding from):

- Department of Commerce (DoC) - Manufacturing efforts: $39B
- **DoC – R&D: $11B**
- CHIPS for America Workforce and Education Fund: $200M
- **CHIPS for America Defense Fund (DoD): $2B**
- CHIPS for America International Technology Security and Innovation Fund (Department of Security, DoS): $500M
- Public Wireless Supply Chain Innovation Fund (DoC): $1.5B

The $2B allotted to the DoD under the CHIPS Act 2022 is focused towards specific, not-necessarily RISC-V-based solutions that tackle issues of the ilk of secure edge computing, 5G/6G technology, hardware and software support for AI/ML, disruptive technologies, etc.

Division B pertaining research and innovation efforts of the overall CHIPS & Science also contemplates some RISC-V-compatible funding avenues. The monetary specifics were not included in the presentation, they were introduced more as collaboration and concession opportunities from different departments:

- National Science Foundation (NSF)
- DoC Technology Hubs: 20 geographically-distributed centers of development and research
- National Institute of Standards and Technology (NIST) authorizations
- National Aeronautics and Space Administration (NASA) authorizations

- Security research for the protection of federal investments in national (US) R&D (DoE)
- Additional DoE Science and Innovation provisions

Another funding avenue still pending approval by the US Congress exists concerning what they label as Trusted & Assured Microelectronics (T&AM), with a budget of $955M. US-based trusted foundry services are hard to come by, and the need for advanced nodes has become a concern and an impediment to the DoD's innovation attempts and ASIC development. So, the DoD aims to reduce barriers to emply mainstream electronics capabilities, while ensuring and protecting critical defense technologies. The ultimate goal is to incorporate ASIC and SoC capabilities offered by the major commercial industry players while maintaining the integrity and security of defense systems to achieve more rapid modernization while reducing the size and increasing performance of DoD systems.

3.4. How can RISC-V help?

SiFive proposes tight cooperation with governmental agencies, the Defense Industrial Base and strategic allies relevant to US national interests. By leveraging the need and desire for asymmetric technological advancements in the compute spectrum of any project, RISC-V can get the funding and financial backing it needs to reach the standard status it has the potential of fullfilling, not only as an open-source ISA but as an integral part of the strategic outline of the USA's presently-shifting priorities moving forward. By joining and collaborating in consortiums and technology hubs with industry leaders such as Lockheed Martin or Raytheon or IBM, the RISC-V ecosystem can be expanded to the next level, but executing this vision will require the collaboration of academia, industry and allied countries, and will require sustained investment over many years. SiFive proposes stablishing strategic partnerships and collaborations at the fab and manufacturing levels, at the software tools level, at the ecosystem level, at the DIB level, concerning international allies and at the academic level.

4. Toolchains tutorial

Speaker: Christoph Müllner – VRULL

The tutorial talk was divided into four major parts:

- Part 1: an explanation of RISC-V international, the mission of the various technical groups within it and their respective specifications around the ISA
- Part 2: an exploration of toolchain concepts, terminology and their various components
- Part 3: hands-on session, building various RISC-V toolchains
- Part 4: second hands-on session, to do with debugging in emulated environments

This part of the report focuses on parts 2 and 3.

4.1. Toolchain concepts, terminology and components

Toolchains are an essential part of the SW ecosystem, they provide all the required tools to enable SW development (in this case, around the RISC-V ISA): think of compilers, linkers and assemblers. Toolchains also bring the fundamental parts of the execution environment, such as language runtimes for C/C++. RISC-V is a versatile and modular ISA, which brings opportunities and challenges for the SW ecosystem, challenges and opportunities addressed both by the toolchain ecosystem. Within the technical groups of RISC-V international, some are of special interest regarding toolchain maintenance and development:

- The Toolchain Special-Interest Group: it develops strategies, identifies gaps and prioritizes tasks for the SW development environment used to build the RISC-V SW ecosystem. It is governed by the Applications & Tools Horizontal Comitees.
- psABI Task Group: currently working on the next psABI specification release.
- Vector intrinsics Task Group: recently created within RISC-V international, it focuses on standardizing the vector intrinsics of the ISA.
- Other community-driven documents:
  - C-API: extensions of the C language (test macros, function attributes, specifics)
  - Toolchain conventions: command line arguments of tools and compilers
  - ASM manual: ASM language extensions

The RISC-V toolchain community interacts and overlaps with many other SW communities, where there is ample room for collaboration and interaction between, often, not-disjointed communities or projects such as GNU’s Binutils, GCC, LLVM, the GNU C library, QEMU, GNU Debugger, Linux as a whole, etc.

The RISC-V toolchain is composed of several types of components, host executables and target executables/libraries among them. The host executables include the compiler/s, the assembler/s for object-file assembly, linker/s to bind multiple object files to a binary/library and more general tools such as objdump or debuggers. Target executable/libraries can include loaders (dynamic linkers), programming language runtimes, support libraries (address sanitizer, openssl, etc.) and header files (stdio.h, stdint.h, iostream and such).

There are also several toolchain types in the RISC-V ecosystem to cater to the needs of various computing fields and platforms. For example, the ELF toolchain (the “bare metal” toolchain) specifically targets embedded systems without feature-rich operating systems. The ELF toolchain makes a point to reduce the “size” of code and, as such, many aspects of it as streamlined to optimize for code-size in exchange for other “fancier” features, such as removing dynamic linking in the interest of low memory footprint of programs and libraries. Some examples are newlib, musl or picolib. The Linux toolchain, on the other hand, targets application support and development within Linux. The Linux toolchain does not concern itself with code-size, and can afford to employ a feature-rich C runtime environment with support for dynamic linking and lots of C extensions (glibc, POSIX APIs, etc.). There are other types of toolchains, but these two are some of the more popular.

There is also a relevant use case regarding RISC-V, which is cross-compiling. When cross-compiling, three types of machines can be distinguished: the build machine (the machine the toolchain is built on), the host machine (the machine using the toolchain later) and the target machine (the machine the toolchain generates code for). Theoretically, one could work on an x86 (build) machine to build a cross-compiler that will be executed on an Aarch64 (host) server machine to build RV32I (target) binaries. The description of machines and toolchain tools is based on machine triplets of the type machine-vendor-os, for example “x86_64-unknown-elf”, “riscv64-none-elf” or “riscv64-unknown-linux-gnu”. If the type of machine is the same for the build, host and target machines, this is called a native toolchain. A cross toolchain implies that build and host are the same but the target machine differs, a cross-native toolchain host and target machines are the same but the build machine isn’t and a “Canadian cross toolchain” has all three machines of different types. Cross toolchains are the most common.

Building a cross toolchain typically implies the following steps:

- Build the assembler and linker (executed on host machine but targeting RISC-V-specific code)
- Build the first (cross-)compiler (executed on host machine but targeting RISC-V-specific code)
- The (cross-)compiler is used to build the C runtime (linked on host machine, executed on target machine)
- Build second (cross-)compiler, such that it has a complete C runtime available now
- Build other language runtimes (linked on host machine, executed on target machine)
- Build other additional libraries or tools such as debuggers (executed on host machine, targeting RISC-V)
- At this point, a “cross-”toolchain is available, and a native toolchain can be built on top of it:
  - Compile/cross-compile assembler/linker (executed on target machine)
  - Compile/cross-compile compiler (executed on target machine)
  - Compile/cross-compile C runtime (executed on target machine)
  - Other runtimes, debuggers...

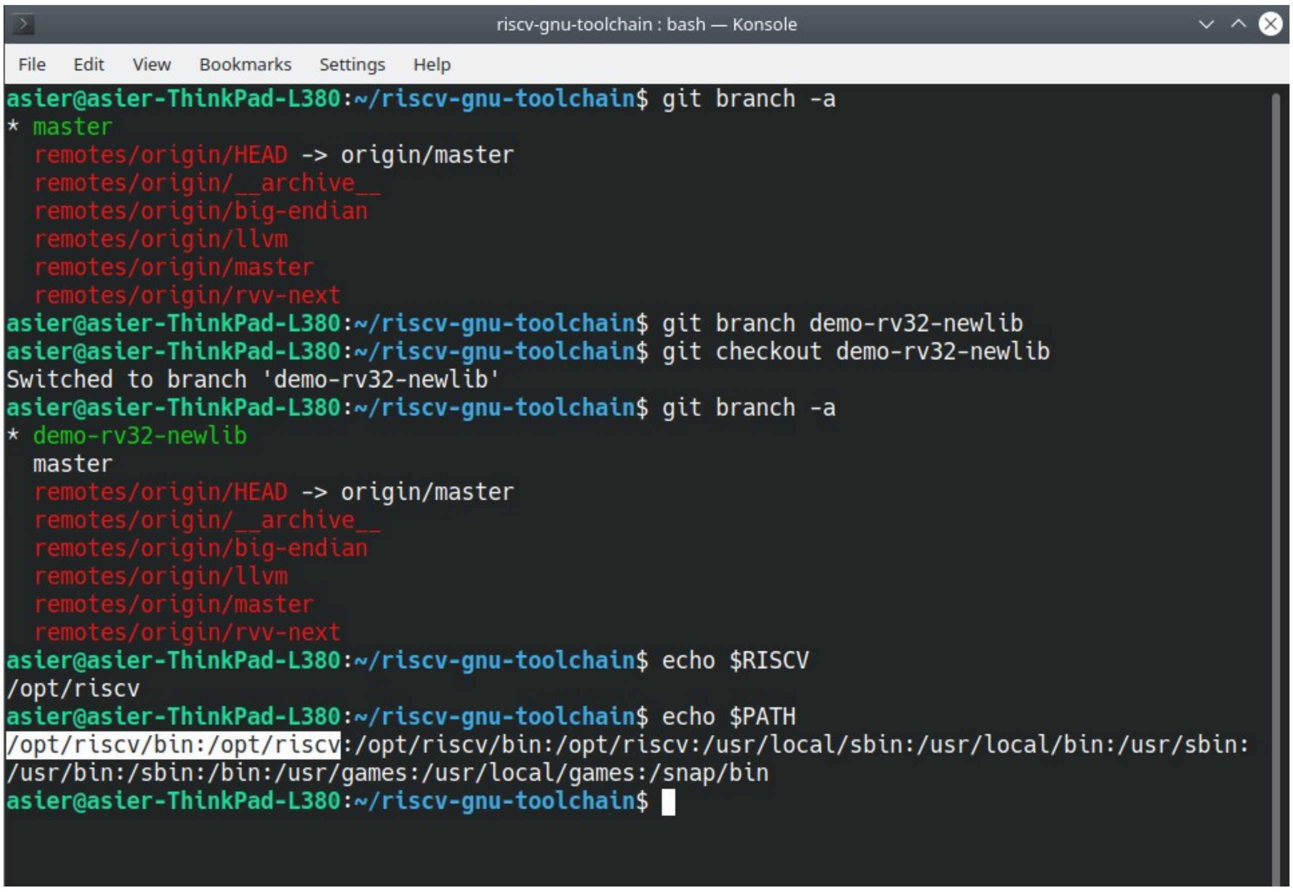
Another relevant concept pertaining toolchains is that of the sysroot. The sysroot is the collection of header files and libraries of the target machine that are required for cross-compilation on the host machine and required for execution on the target machine. More detailedly, out of all components resulting from the above compilation steps, some are linked on the host machine but (1) are actually never executed there and (2) are instead meant to be executed on the target machine. Those componenes are labelled as “linked on host machine, executed on target machine”. They are a collection of header and library files shared among the cross compiler in the host machine and the target system.

There are a bunch of options for building a RISC-V toolchain. Some popular options are the Linux From Scratch (LFS) project or toolchain builder projects like crosstool-NG or riscv-gnu-toolchain. The first aims to build a complete Linux system from scratch from the command line, while the others are more specific toolchain builder projects supporting many target machine types, native tools and provide more flexibility of sysroot components. The most popular toolchain builder for all things RISC-V is the riscv-gnu-toolchain project and is a good choice for RISC-V development as a whole. It currently supports the GNU toolchain plus QEMU only, but there exists a pull request for LLVM support. One of the more useful aspects of the toolchain project is the execution of the GCC regression test suite.

4.2. Toolchain hands-on session

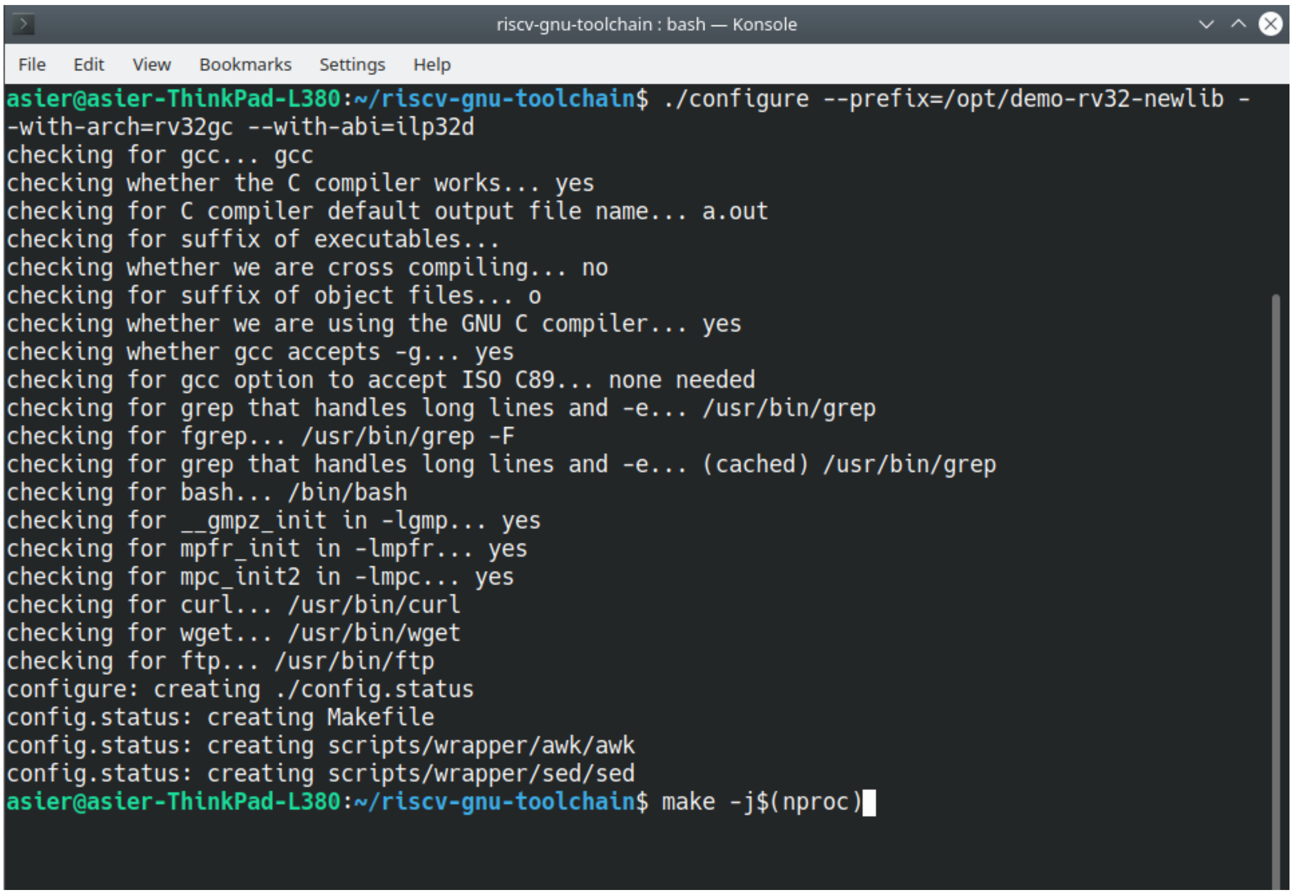
The hands-on session revolved around the various uses one can devote the riscv-gnu-toolchain project to. Specifically, the tutorial session explored the compilation and build processes of a newlib (elf bare-metal) 32-bit toolchain, a 64-bit linux-capable version of the toolchain along with QEMU to execute binaries and actual debugging of faulty code with GDB and QEMU with the 64-bit toolchain.

After cloning the repository, create a branch for the type of toolchain to be built, for keepsake. The RISCV env variable should also exist in your path, just in case and provided one wishes to follow other tutorials from other RISC-V-related projects. This is done for both toolchains visited in the hands-on session.


riscv-gnu-toolchain : bash — Konsole
File Edit View Bookmarks Settings Help

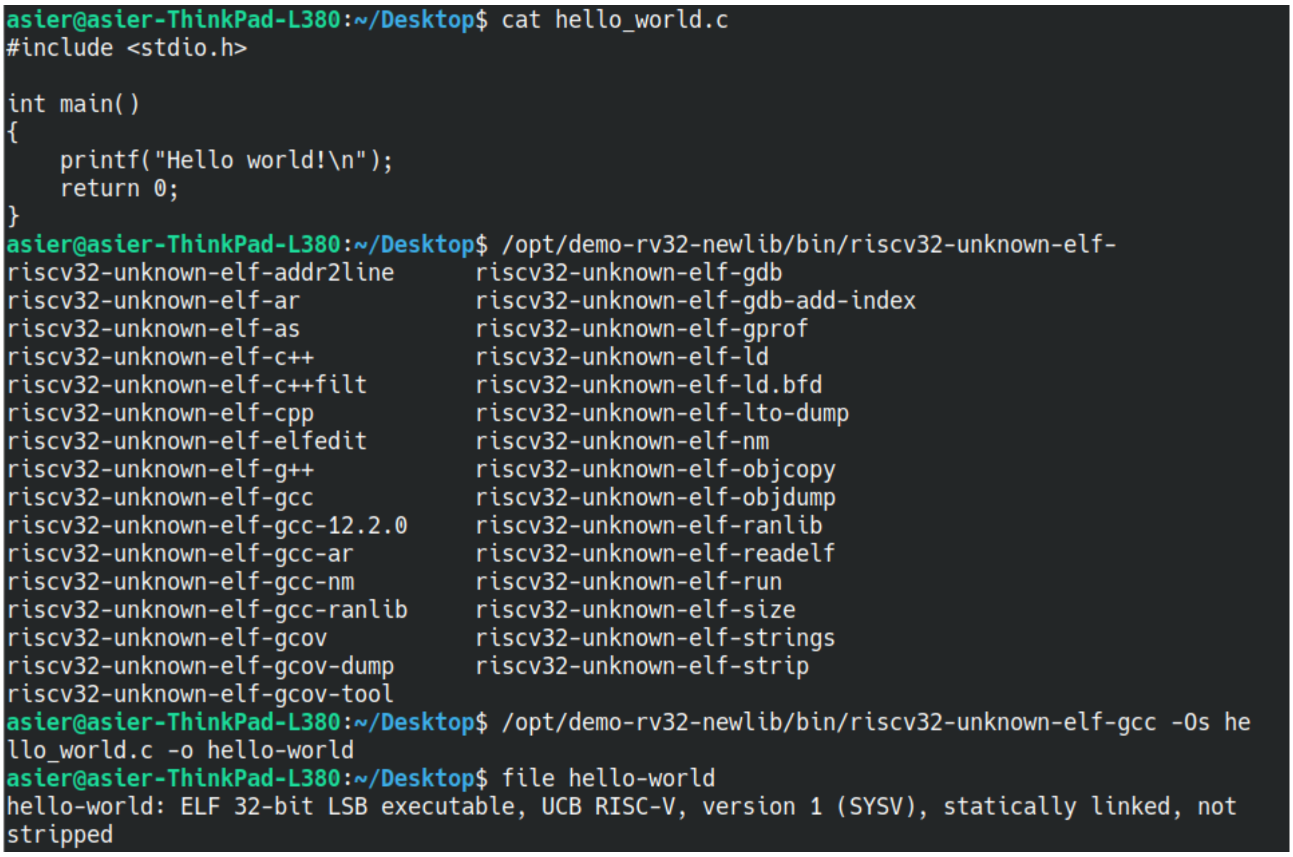
```
asier@asier-ThinkPad-L380:~/riscv-gnu-toolchain$ git branch -a
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/__archive__
  remotes/origin/big-endian
  remotes/origin/llvm
  remotes/origin/master
  remotes/origin/rvv-next
asier@asier-ThinkPad-L380:~/riscv-gnu-toolchain$ git branch demo-rv32-newlib
asier@asier-ThinkPad-L380:~/riscv-gnu-toolchain$ git checkout demo-rv32-newlib
Switched to branch 'demo-rv32-newlib'
asier@asier-ThinkPad-L380:~/riscv-gnu-toolchain$ git branch -a
* demo-rv32-newlib
  master
  remotes/origin/HEAD -> origin/master
  remotes/origin/__archive__
  remotes/origin/big-endian
  remotes/origin/llvm
  remotes/origin/master
  remotes/origin/rvv-next
asier@asier-ThinkPad-L380:~/riscv-gnu-toolchain$ echo $RISCV
/opt/riscv
asier@asier-ThinkPad-L380:~/riscv-gnu-toolchain$ echo $PATH
/opt/riscv/bin:/opt/riscv:/opt/riscv/bin:/opt/riscv:/usr/local/sbin:/usr/local/bin:/usr/sbin:
/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
asier@asier-ThinkPad-L380:~/riscv-gnu-toolchain$ 
```

The configuring and compilation of a 32-bit, bare-metal RISC-V toolchain looks as depicted below. It takes around one to two hours to finish, depending on your machine. Some other parameters are also added to notice the details in the compiled .c programs later. After compilation, one should “echo $?” to check the status of the compilation process, expecting a zero code for no errors.


riscv-gnu-toolchain : bash — Konsole
File Edit View Bookmarks Settings Help

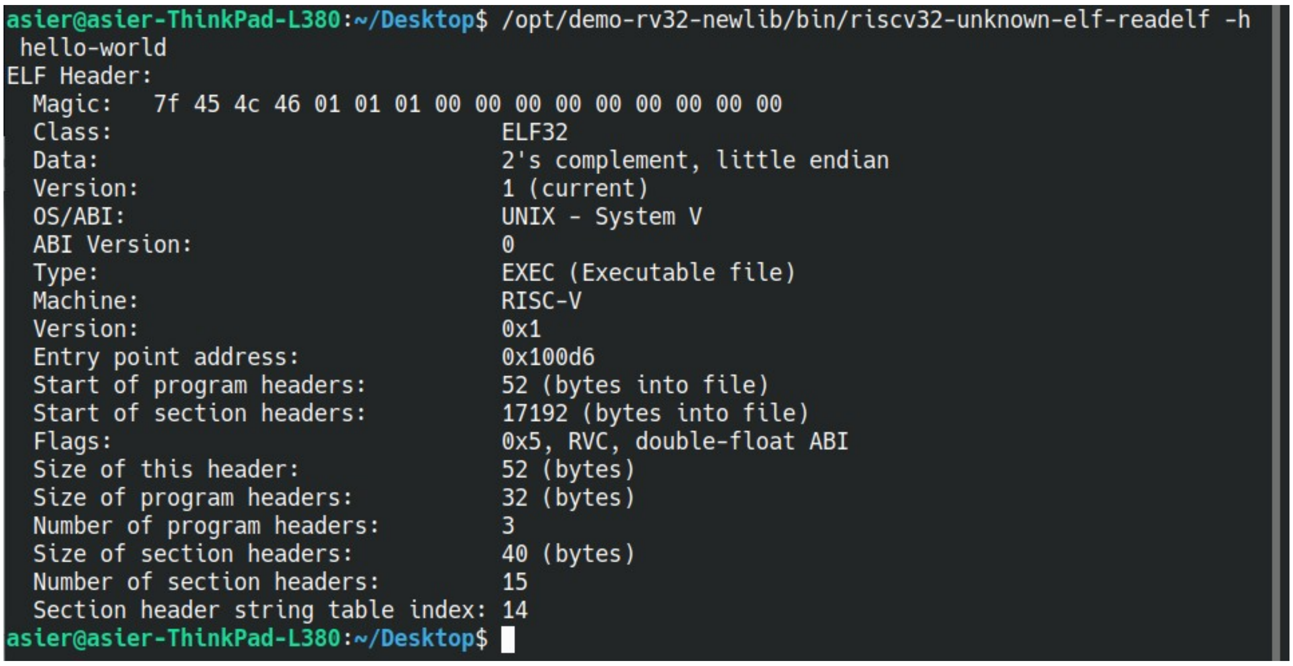
```
asier@asier-ThinkPad-L380:~/riscv-gnu-toolchain$ ./configure --prefix=/opt/demo-rv32-newlib -
-with-arch=rv32gc --with-abi=ilp32d
checking for gcc... gcc
checking whether the C compiler works... yes
checking for C compiler default output file name... a.out
checking for suffix of executables...
checking whether we are cross compiling... no
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ISO C89... none needed
checking for grep that handles long lines and -e... /usr/bin/grep
checking for fgrep... /usr/bin/grep -F
checking for grep that handles long lines and -e... (cached) /usr/bin/grep
checking for bash... /bin/bash
checking for __gmpz_init in -lgmp... yes
checking for mpfr_init in -lmpfr... yes
checking for mpc_init2 in -lmpc... yes
checking for curl... /usr/bin/curl
checking for wget... /usr/bin/wget
checking for ftp... /usr/bin/ftp
configure: creating ./config.status
config.status: creating Makefile
config.status: creating scripts/wrapper/awk/awk
config.status: creating scripts/wrapper/sed/sed
asier@asier-ThinkPad-L380:~/riscv-gnu-toolchain$ make -j$(nproc)
```

The compilation of the toolchain has created many tools and executables in the /opt/demo-rv32-newlib/ directory. Among them, one can find the GCC compiler, which outputs 32-bit, statically-linked binaries (this is a property of the bare-metal newlib toolchain, optimizing for code size).



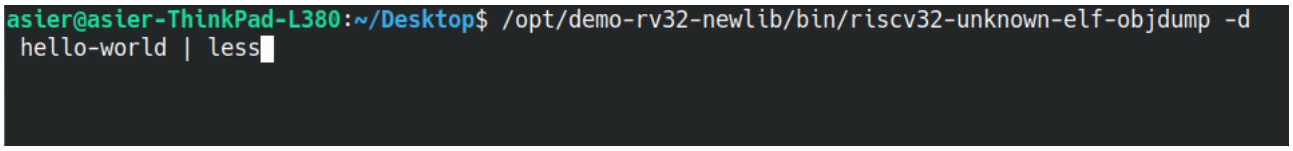
```
asier@asier-ThinkPad-L380:~/Desktop$ cat hello_world.c
#include <stdio.h>

int main()
{
    printf("Hello world!\n");
    return 0;
}
asier@asier-ThinkPad-L380:~/Desktop$ /opt/demo-rv32-newlib/bin/riscv32-unknown-elf-
riscv32-unknown-elf-addr2line      riscv32-unknown-elf-gdb
riscv32-unknown-elf-ar             riscv32-unknown-elf-gdb-add-index
riscv32-unknown-elf-as             riscv32-unknown-elf-gprof
riscv32-unknown-elf-c++            riscv32-unknown-elf-ld
riscv32-unknown-elf-c++filt        riscv32-unknown-elf-ld.bfd
riscv32-unknown-elf-cpp            riscv32-unknown-elf-lto-dump
riscv32-unknown-elf-elfedit        riscv32-unknown-elf-nm
riscv32-unknown-elf-g++            riscv32-unknown-elf-objcopy
riscv32-unknown-elf-gcc            riscv32-unknown-elf-objdump
riscv32-unknown-elf-gcc-12.2.0     riscv32-unknown-elf-ranlib
riscv32-unknown-elf-gcc-ar         riscv32-unknown-elf-readelf
riscv32-unknown-elf-gcc-nm         riscv32-unknown-elf-run
riscv32-unknown-elf-gcc-ranlib     riscv32-unknown-elf-size
riscv32-unknown-elf-gcov           riscv32-unknown-elf-strings
riscv32-unknown-elf-gcov-dump      riscv32-unknown-elf-strip
riscv32-unknown-elf-gcov-tool
asier@asier-ThinkPad-L380:~/Desktop$ /opt/demo-rv32-newlib/bin/riscv32-unknown-elf-gcc -Os he
llo_world.c -o hello-world
asier@asier-ThinkPad-L380:~/Desktop$ file hello-world
hello-world: ELF 32-bit LSB executable, UCB RISC-V, version 1 (SYSV), statically linked, not
stripped
```

Other tools such as the “readelf” utility allows us to gain deeper insight into the binary’s properties.

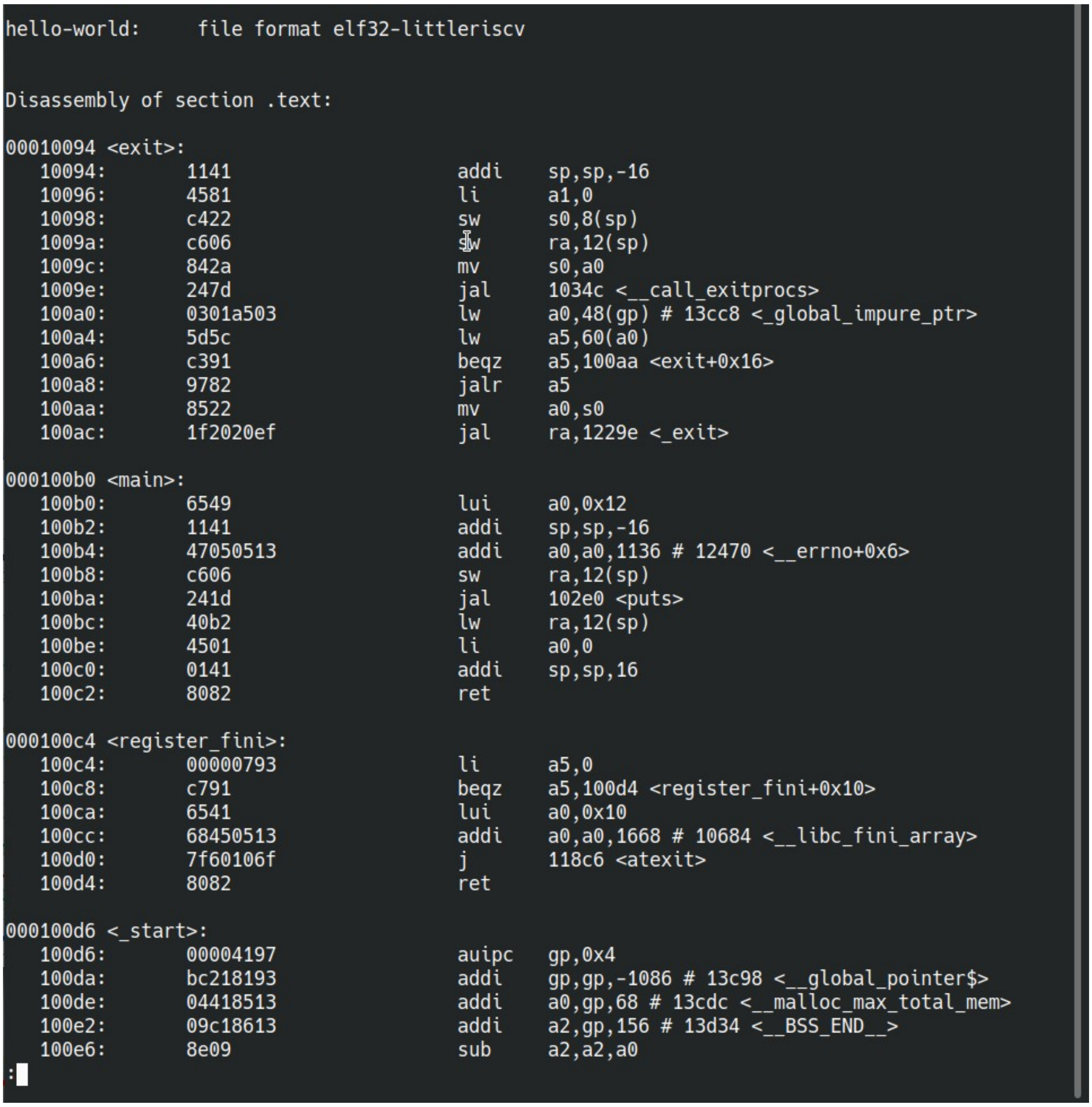


```
asier@asier-ThinkPad-L380:~/Desktop$ /opt/demo-rv32-newlib/bin/riscv32-unknown-elf-readelf -h
 hello-world
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           RISC-V
  Version:                           0x1
  Entry point address:               0x100d6
  Start of program headers:          52 (bytes into file)
  Start of section headers:          17192 (bytes into file)
  Flags:                             0x5, RVC, double-float ABI
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         3
  Size of section headers:           40 (bytes)
  Number of section headers:         15
  Section header string table index: 14
asier@asier-ThinkPad-L380:~/Desktop$ 
```

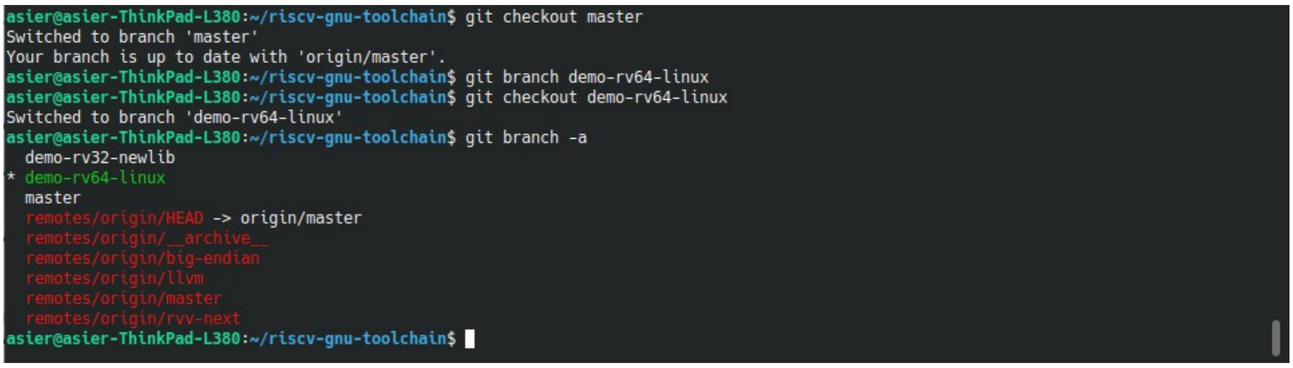
Finally, objdump allows us to inspect the actual assembly code of the executable.

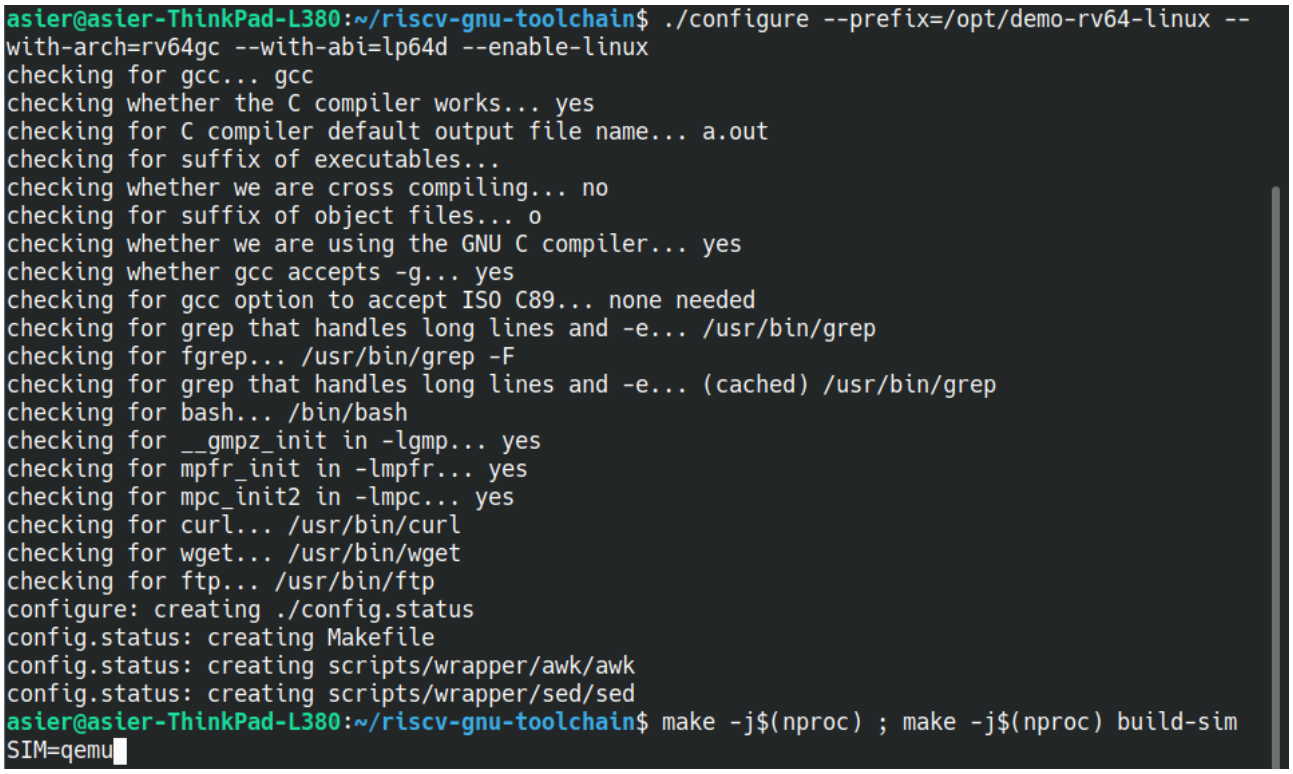


```
asier@asier-ThinkPad-L380:~/Desktop$ /opt/demo-rv32-newlib/bin/riscv32-unknown-elf-objdump -d
 hello-world | less
```

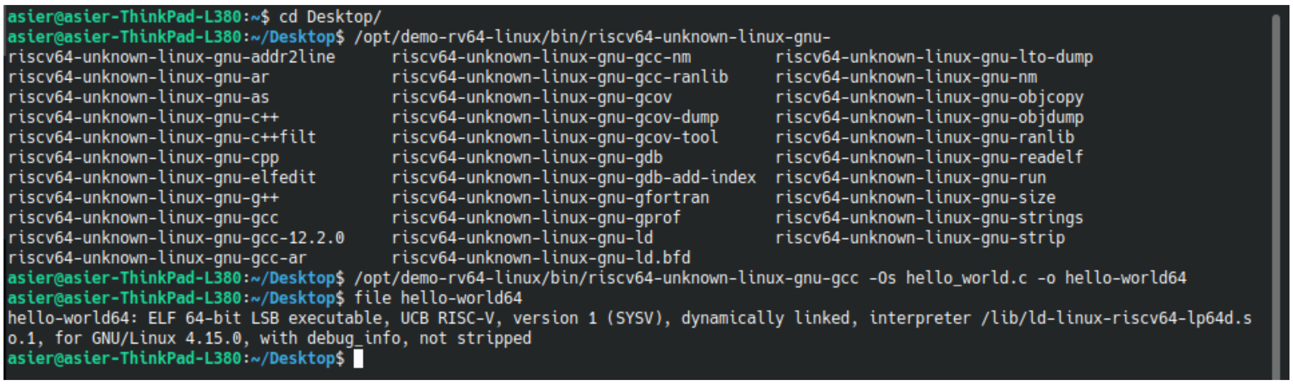

```
hello-world:     file format elf32-littleriscv


Disassembly of section .text:

00010094 <exit>:
   10094:       1141                    addi    sp,sp,-16
   10096:       4581                    li      a1,0
   10098:       c422                    sw      s0,8(sp)
   1009a:       c606                    sw      ra,12(sp)
   1009c:       842a                    mv      s0,a0
   1009e:       247d                    jal     1034c <__call_exitprocs>
   100a0:       0301a503                lw      a0,48(gp) # 13cc8 <_global_impure_ptr>
   100a4:       5d5c                    lw      a5,60(a0)
   100a6:       c391                    beqz    a5,100aa <exit+0x16>
   100a8:       9782                    jalr    a5
   100aa:       8522                    mv      a0,s0
   100ac:       1f2020ef                jal     ra,1229e <_exit>

000100b0 <main>:
   100b0:       6549                    lui     a0,0x12
   100b2:       1141                    addi    sp,sp,-16
   100b4:       47050513                addi    a0,a0,1136 # 12470 <__errno+0x6>
   100b8:       c606                    sw      ra,12(sp)
   100ba:       241d                    jal     102e0 <puts>
   100bc:       40b2                    lw      ra,12(sp)
   100be:       4501                    li      a0,0
   100c0:       0141                    addi    sp,sp,16
   100c2:       8082                    ret

000100c4 <register_fini>:
   100c4:       00000793                li      a5,0
   100c8:       c791                    beqz    a5,100d4 <register_fini+0x10>
   100ca:       6541                    lui     a0,0x10
   100cc:       68450513                addi    a0,a0,1668 # 10684 <__libc_fini_array>
   100d0:       7f60106f                j       118c6 <atexit>
   100d4:       8082                    ret

000100d6 <_start>:
   100d6:       00004197                auipc   gp,0x4
   100da:       bc218193                addi    gp,gp,-1086 # 13c98 <__global_pointer$>
   100de:       04418513                addi    a0,gp,68 # 13cdc <__malloc_max_total_mem>
   100e2:       09c18613                addi    a2,gp,156 # 13d34 <__BSS_END__>
   100e6:       8e09                    sub     a2,a2,a0
:
```

The compilation process of a 64-bit, linux capable RISC-V toolchain requires similar steps as before: creating a branch within the cloned repository, compiling and building the actual toolchain (this time along with QEMU for execution of binaries) and playing around with the compiled hello-world programs. This time, the compiled programs will have fancier features like dynamic linking, for example.


```
asier@asier-ThinkPad-L380:~/riscv-gnu-toolchain$ git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.
asier@asier-ThinkPad-L380:~/riscv-gnu-toolchain$ git branch demo-rv64-linux
asier@asier-ThinkPad-L380:~/riscv-gnu-toolchain$ git checkout demo-rv64-linux
Switched to branch 'demo-rv64-linux'
asier@asier-ThinkPad-L380:~/riscv-gnu-toolchain$ git branch -a
  demo-rv32-newlib
* demo-rv64-linux
  master
  remotes/origin/HEAD -> origin/master
  remotes/origin/__archive__
  remotes/origin/big-endian
  remotes/origin/llvm
  remotes/origin/master
  remotes/origin/rvv-next
asier@asier-ThinkPad-L380:~/riscv-gnu-toolchain$
```

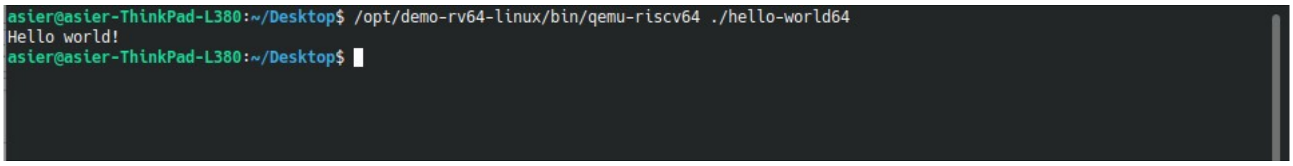


```
asier@asier-ThinkPad-L380:~/riscv-gnu-toolchain$ ./configure --prefix=/opt/demo-rv64-linux --
with-arch=rv64gc --with-abi=lp64d --enable-linux
checking for gcc... gcc
checking whether the C compiler works... yes
checking for C compiler default output file name... a.out
checking for suffix of executables...
checking whether we are cross compiling... no
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ISO C89... none needed
checking for grep that handles long lines and -e... /usr/bin/grep
checking for fgrep... /usr/bin/grep -F
checking for grep that handles long lines and -e... (cached) /usr/bin/grep
checking for bash... /bin/bash
checking for __gmpz_init in -lgmp... yes
checking for mpfr_init in -lmpfr... yes
checking for mpc_init2 in -lmpc... yes
checking for curl... /usr/bin/curl
checking for wget... /usr/bin/wget
checking for ftp... /usr/bin/ftp
configure: creating ./config.status
config.status: creating Makefile
config.status: creating scripts/wrapper/awk/awk
config.status: creating scripts/wrapper/sed/sed
asier@asier-ThinkPad-L380:~/riscv-gnu-toolchain$ make -j$(nproc) ; make -j$(nproc) build-sim
SIM=qemu
```

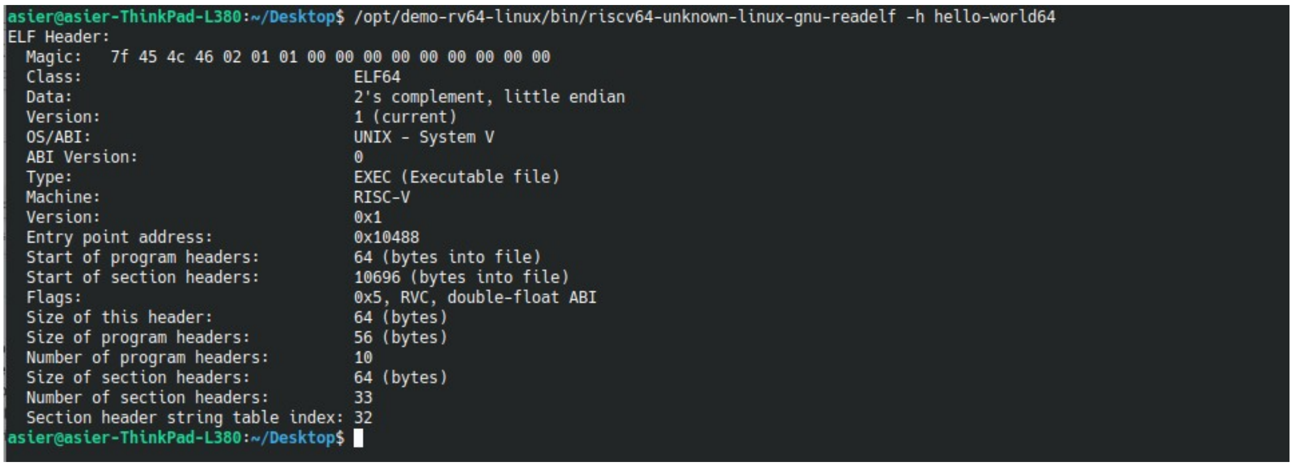


```
asier@asier-ThinkPad-L380:~$ cd Desktop/
asier@asier-ThinkPad-L380:~/Desktop$ /opt/demo-rv64-linux/bin/riscv64-unknown-linux-gnu-
riscv64-unknown-linux-gnu-addr2line      riscv64-unknown-linux-gnu-gcc-nm         riscv64-unknown-linux-gnu-lto-dump
riscv64-unknown-linux-gnu-ar             riscv64-unknown-linux-gnu-gcc-ranlib     riscv64-unknown-linux-gnu-nm
riscv64-unknown-linux-gnu-as             riscv64-unknown-linux-gnu-gcov           riscv64-unknown-linux-gnu-objcopy
riscv64-unknown-linux-gnu-c++            riscv64-unknown-linux-gnu-gcov-dump      riscv64-unknown-linux-gnu-objdump
riscv64-unknown-linux-gnu-c++filt        riscv64-unknown-linux-gnu-gcov-tool      riscv64-unknown-linux-gnu-ranlib
riscv64-unknown-linux-gnu-cpp            riscv64-unknown-linux-gnu-gdb            riscv64-unknown-linux-gnu-readelf
riscv64-unknown-linux-gnu-elfedit        riscv64-unknown-linux-gnu-gdb-add-index  riscv64-unknown-linux-gnu-run
riscv64-unknown-linux-gnu-g++            riscv64-unknown-linux-gnu-gfortran       riscv64-unknown-linux-gnu-size
riscv64-unknown-linux-gnu-gcc            riscv64-unknown-linux-gnu-gprof          riscv64-unknown-linux-gnu-strings
riscv64-unknown-linux-gnu-gcc-12.2.0     riscv64-unknown-linux-gnu-ld             riscv64-unknown-linux-gnu-strip
riscv64-unknown-linux-gnu-gcc-ar         riscv64-unknown-linux-gnu-ld.bfd
asier@asier-ThinkPad-L380:~/Desktop$ /opt/demo-rv64-linux/bin/riscv64-unknown-linux-gnu-gcc -Os hello_world.c -o hello-world64
asier@asier-ThinkPad-L380:~/Desktop$ file hello-world64
hello-world64: ELF 64-bit LSB executable, UCB RISC-V, version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux-riscv64-lp64d.s
o.1, for GNU/Linux 4.15.0, with debug_info, not stripped
asier@asier-ThinkPad-L380:~/Desktop$
```

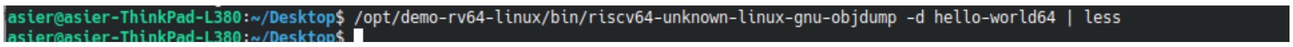
QEMU now allows for the actual execution of the hello-world program.



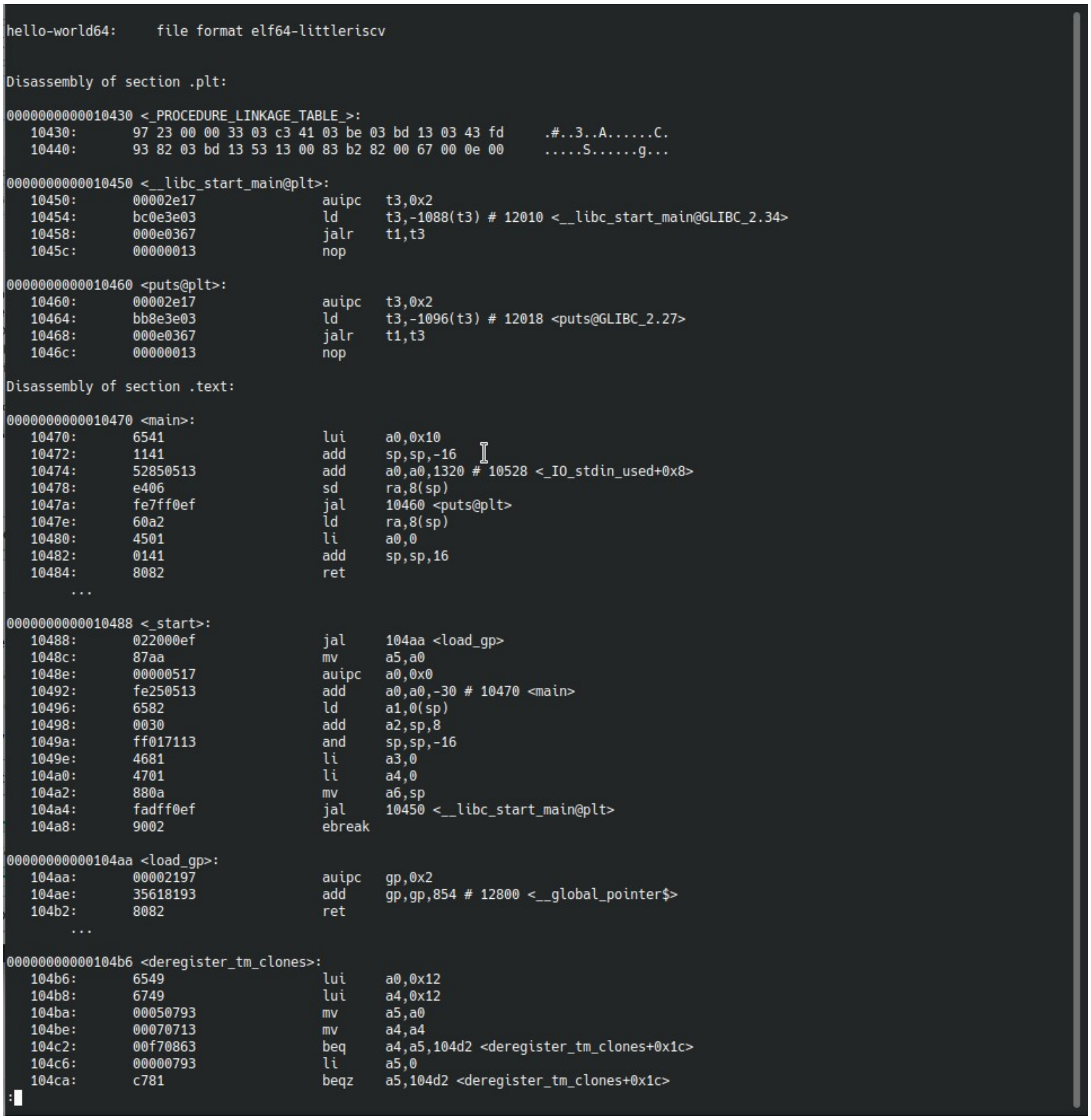
```
asier@asier-ThinkPad-L380:~/Desktop$ /opt/demo-rv64-linux/bin/qemu-riscv64 ./hello-world64
Hello world!
asier@asier-ThinkPad-L380:~/Desktop$
```



```
asier@asier-ThinkPad-L380:~/Desktop$ /opt/demo-rv64-linux/bin/riscv64-unknown-linux-gnu-readelf -h hello-world64
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           RISC-V
  Version:                           0x1
  Entry point address:               0x10488
  Start of program headers:          64 (bytes into file)
  Start of section headers:          10696 (bytes into file)
  Flags:                             0x5, RVC, double-float ABI
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         10
  Size of section headers:           64 (bytes)
  Number of section headers:         33
  Section header string table index: 32
asier@asier-ThinkPad-L380:~/Desktop$
```

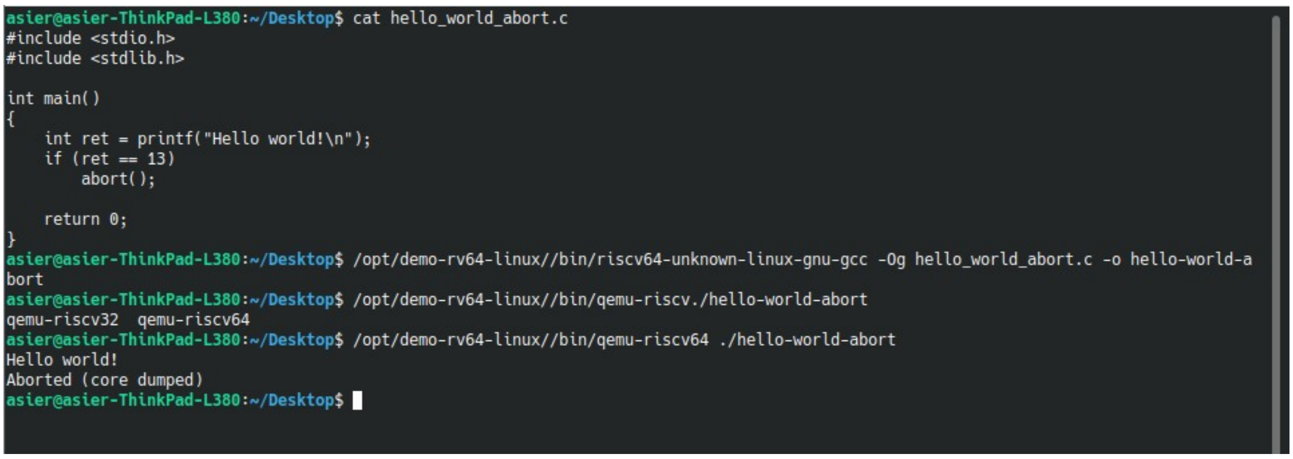


```
asier@asier-ThinkPad-L380:~/Desktop$ /opt/demo-rv64-linux/bin/riscv64-unknown-linux-gnu-objdump -d hello-world64 | less
asier@asier-ThinkPad-L380:~/Desktop$
```



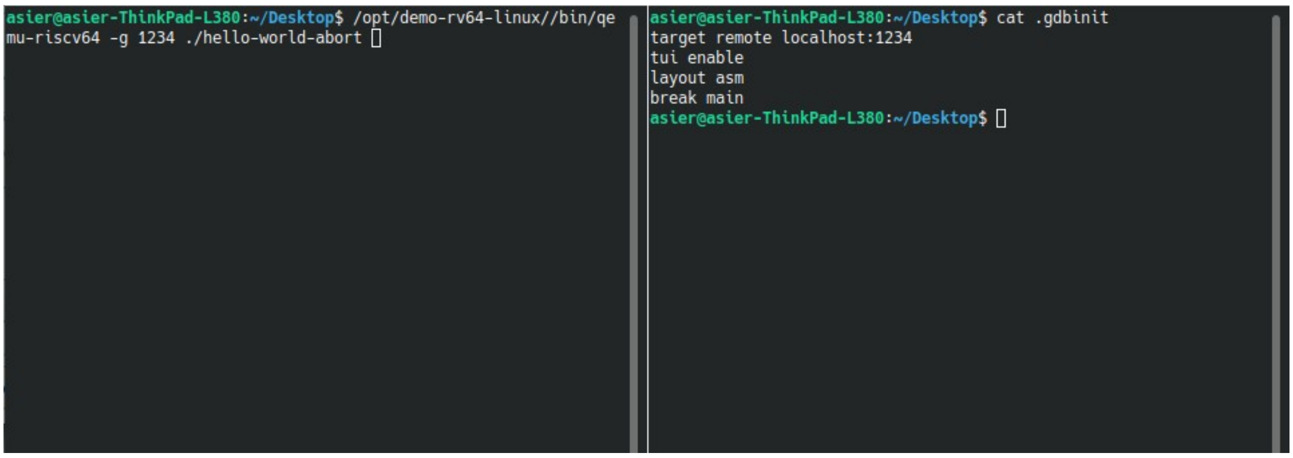
```

hello-world64:     file format elf64-littleriscv


Disassembly of section .plt:

0000000000010430 <_PROCEDURE_LINKAGE_TABLE_>:
   10430:       97 23 00 00 33 03 c3 41 03 be 03 bd 13 03 43 fd     .#..3..A......C.
   10440:       93 82 03 bd 13 53 13 00 83 b2 82 00 67 00 0e 00     .....S......g...

0000000000010450 <__libc_start_main@plt>:
   10450:       00002e17                auipc   t3,0x2
   10454:       bc0e3e03                ld      t3,-1088(t3) # 12010 <__libc_start_main@GLIBC_2.34>
   10458:       000e0367                jalr    t1,t3
   1045c:       00000013                nop

0000000000010460 <puts@plt>:
   10460:       00002e17                auipc   t3,0x2
   10464:       bb8e3e03                ld      t3,-1096(t3) # 12018 <puts@GLIBC_2.27>
   10468:       000e0367                jalr    t1,t3
   1046c:       00000013                nop

Disassembly of section .text:

0000000000010470 <main>:
   10470:       6541                    lui     a0,0x10
   10472:       1141                    add     sp,sp,-16
   10474:       52850513                add     a0,a0,1320 # 10528 <_IO_stdin_used+0x8>
   10478:       e406                    sd      ra,8(sp)
   1047a:       fe7ff0ef                jal     10460 <puts@plt>
   1047e:       60a2                    ld      ra,8(sp)
   10480:       4501                    li      a0,0
   10482:       0141                    add     sp,sp,16
   10484:       8082                    ret
        ...

0000000000010488 <_start>:
   10488:       022000ef                jal     104aa <load_gp>
   1048c:       87aa                    mv      a5,a0
   1048e:       00000517                auipc   a0,0x0
   10492:       fe250513                add     a0,a0,-30 # 10470 <main>
   10496:       6582                    ld      a1,0(sp)
   10498:       0030                    add     a2,sp,8
   1049a:       ff017113                and     sp,sp,-16
   1049e:       4681                    li      a3,0
   104a0:       4701                    li      a4,0
   104a2:       880a                    mv      a6,sp
   104a4:       fadff0ef                jal     10450 <__libc_start_main@plt>
   104a8:       9002                    ebreak

00000000000104aa <load_gp>:
   104aa:       00002197                auipc   gp,0x2
   104ae:       35618193                add     gp,gp,854 # 12800 <__global_pointer$>
   104b2:       8082                    ret
        ...

00000000000104b6 <deregister_tm_clones>:
   104b6:       6549                    lui     a0,0x12
   104b8:       6749                    lui     a4,0x12
   104ba:       00050793                mv      a5,a0
   104be:       00070713                mv      a4,a4
   104c2:       00f70863                beq     a4,a5,104d2 <deregister_tm_clones+0x1c>
   104c6:       00000793                li      a5,0
   104ca:       c781                    beqz    a5,104d2 <deregister_tm_clones+0x1c>
:
```

Let’s take a look now at how plain GDB works on a faulty hello-world program.



```
asier@asier-ThinkPad-L380:~/Desktop$ cat hello_world_abort.c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int ret = printf("Hello world!\n");
    if (ret == 13)
        abort();

    return 0;
}
asier@asier-ThinkPad-L380:~/Desktop$ /opt/demo-rv64-linux//bin/riscv64-unknown-linux-gnu-gcc -Og hello_world_abort.c -o hello-world-a
bort
asier@asier-ThinkPad-L380:~/Desktop$ /opt/demo-rv64-linux//bin/qemu-riscv./hello-world-abort
qemu-riscv32  qemu-riscv64
asier@asier-ThinkPad-L380:~/Desktop$ /opt/demo-rv64-linux//bin/qemu-riscv64 ./hello-world-abort
Hello world!
Aborted (core dumped)
asier@asier-ThinkPad-L380:~/Desktop$
```

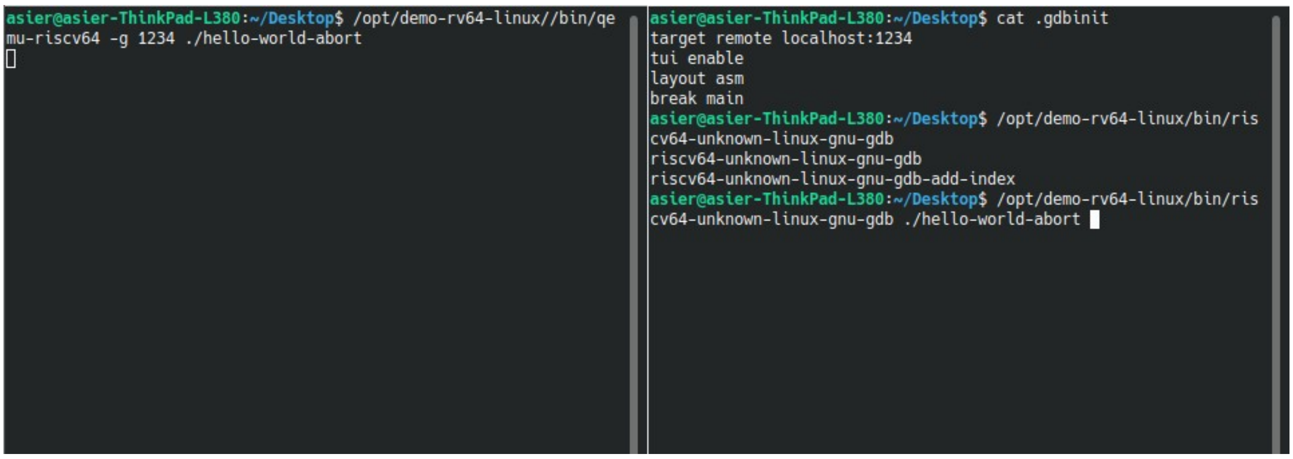
For debugging the .c program, it is necessary to “tell” GDB how to execute it in the first place. As such, QEMU can open a server-like process for GDB to inspect its execution step by step. GDB requires some configuration first, however, as to the specifics of the debugging process, such as the port of the process where the executable is waiting to be inspected or where the breakpoint should be placed (see “break main”, indicating that a breakpoint is placed by default on the main function).



```
asier@asier-ThinkPad-L380:~/Desktop$ /opt/demo-rv64-linux//bin/qe
mu-riscv64 -g 1234 ./hello-world-abort
```

```
asier@asier-ThinkPad-L380:~/Desktop$ cat .gdbinit
target remote localhost:1234
tui enable
layout asm
break main
asier@asier-ThinkPad-L380:~/Desktop$
```

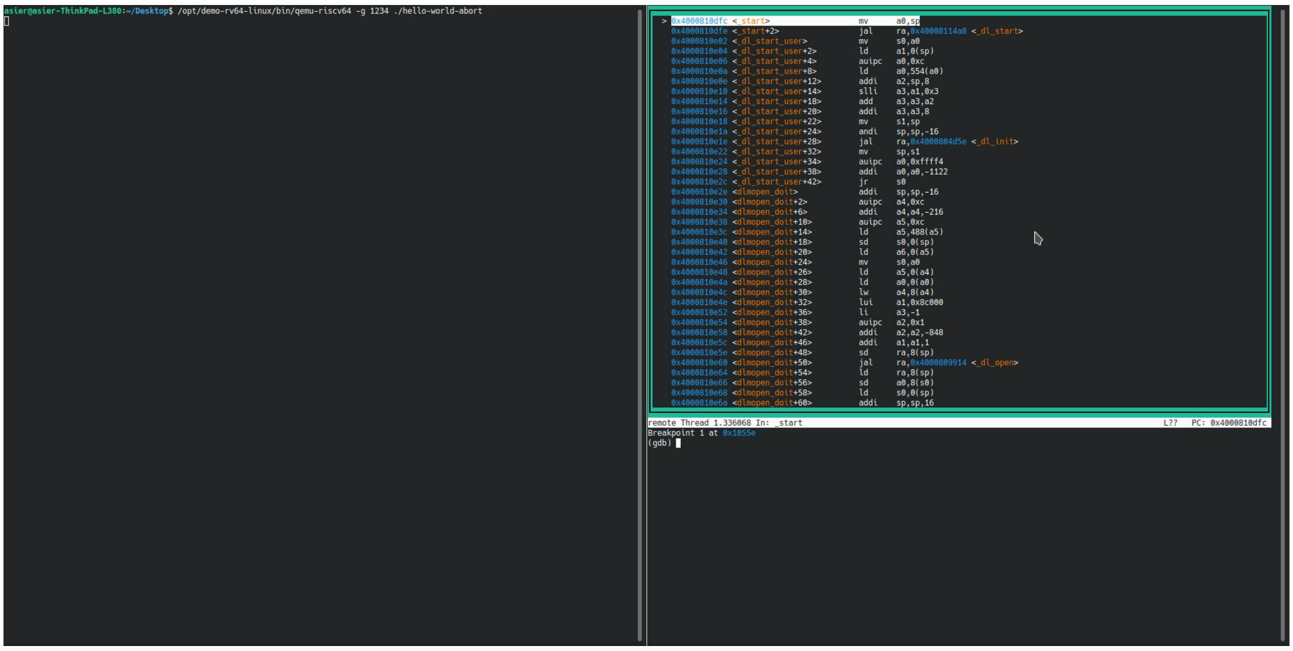
After executing the QEMU “debugging server”, GDB can inspect the .c program.



```
asier@asier-ThinkPad-L380:~/Desktop$ /opt/demo-rv64-linux//bin/qe
mu-riscv64 -g 1234 ./hello-world-abort
```

```
asier@asier-ThinkPad-L380:~/Desktop$ cat .gdbinit
target remote localhost:1234
tui enable
layout asm
break main
asier@asier-ThinkPad-L380:~/Desktop$ /opt/demo-rv64-linux/bin/ris
cv64-unknown-linux-gnu-gdb
riscv64-unknown-linux-gnu-gdb
riscv64-unknown-linux-gnu-gdb-add-index
asier@asier-ThinkPad-L380:~/Desktop$ /opt/demo-rv64-linux/bin/ris
cv64-unknown-linux-gnu-gdb ./hello-world-abort
```

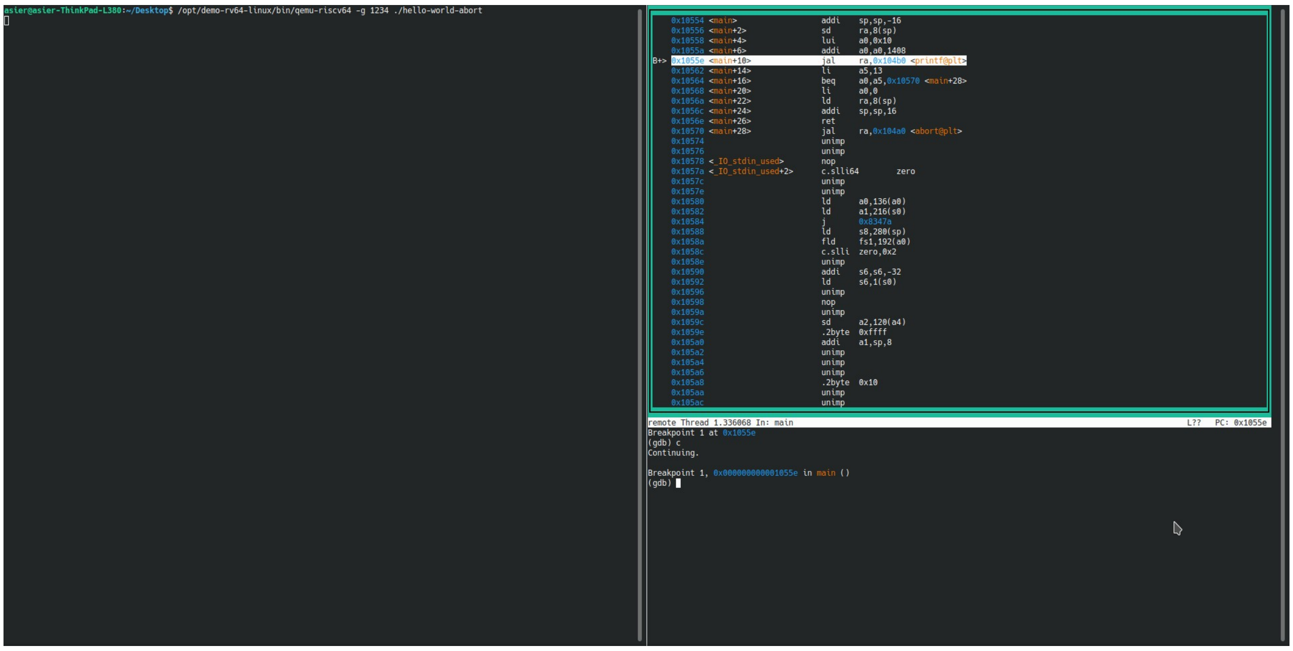
The console on the right shows GDB running, having stopped at the start of the program’s execution waiting for further input.



```
asier@asier-ThinkPad-L380:~/Desktop$ /opt/demo-rv64-linux/bin/qemu-riscv64 -g 1234 ./hello-world-abort
```

```
  > 0x4000810dfc <_start>                 mv      a0,sp
    0x4000810dfe <_start+2>               jal     ra,0x40008114a8 <_dl_start>
    0x4000810e02 <_dl_start_user>         mv      s0,a0
    0x4000810e04 <_dl_start_user+2>       ld      a1,0(sp)
    0x4000810e06 <_dl_start_user+4>       auipc   a0,0xc
    0x4000810e0a <_dl_start_user+8>       ld      a0,554(a0)
    0x4000810e0e <_dl_start_user+12>      addi    a2,sp,8
    0x4000810e10 <_dl_start_user+14>      slli    a3,a1,0x3
    0x4000810e14 <_dl_start_user+18>      add     a3,a3,a2
    0x4000810e16 <_dl_start_user+20>      addi    a3,a3,8
    0x4000810e18 <_dl_start_user+22>      mv      s1,sp
    0x4000810e1a <_dl_start_user+24>      andi    sp,sp,-16
    0x4000810e1e <_dl_start_user+28>      jal     ra,0x4000804d5e <_dl_init>
    0x4000810e22 <_dl_start_user+32>      mv      sp,s1
    0x4000810e24 <_dl_start_user+34>      auipc   a0,0xffff4
    0x4000810e28 <_dl_start_user+38>      addi    a0,a0,-1122
    0x4000810e2c <_dl_start_user+42>      jr      s0
    0x4000810e2e <dlmopen_doit>           addi    sp,sp,-16
    0x4000810e30 <dlmopen_doit+2>         auipc   a4,0xc
    0x4000810e34 <dlmopen_doit+6>         addi    a4,a4,-216
    0x4000810e38 <dlmopen_doit+10>        auipc   a5,0xc
    0x4000810e3c <dlmopen_doit+14>        ld      a5,488(a5)
    0x4000810e40 <dlmopen_doit+18>        sd      s0,0(sp)
    0x4000810e42 <dlmopen_doit+20>        ld      a6,0(a5)
    0x4000810e46 <dlmopen_doit+24>        mv      s0,a0
    0x4000810e48 <dlmopen_doit+26>        ld      a5,0(a4)
    0x4000810e4a <dlmopen_doit+28>        ld      a0,0(a0)
    0x4000810e4c <dlmopen_doit+30>        lw      a4,8(a4)
    0x4000810e4e <dlmopen_doit+32>        lui     a1,0x8c000
    0x4000810e52 <dlmopen_doit+36>        li      a3,-1
    0x4000810e54 <dlmopen_doit+38>        auipc   a2,0x1
    0x4000810e58 <dlmopen_doit+42>        addi    a2,a2,-848
    0x4000810e5c <dlmopen_doit+46>        addi    a1,a1,1
    0x4000810e5e <dlmopen_doit+48>        sd      ra,8(sp)
    0x4000810e60 <dlmopen_doit+50>        jal     ra,0x4000809914 <_dl_open>
    0x4000810e64 <dlmopen_doit+54>        ld      ra,8(sp)
    0x4000810e66 <dlmopen_doit+56>        sd      a0,8(s0)
    0x4000810e68 <dlmopen_doit+58>        ld      s0,0(sp)
    0x4000810e6a <dlmopen_doit+60>        addi    sp,sp,16
remote Thread 1.336068 In: _start                                          L??   PC: 0x4000810dfc
Breakpoint 1 at 0x1055e
(gdb) 
```

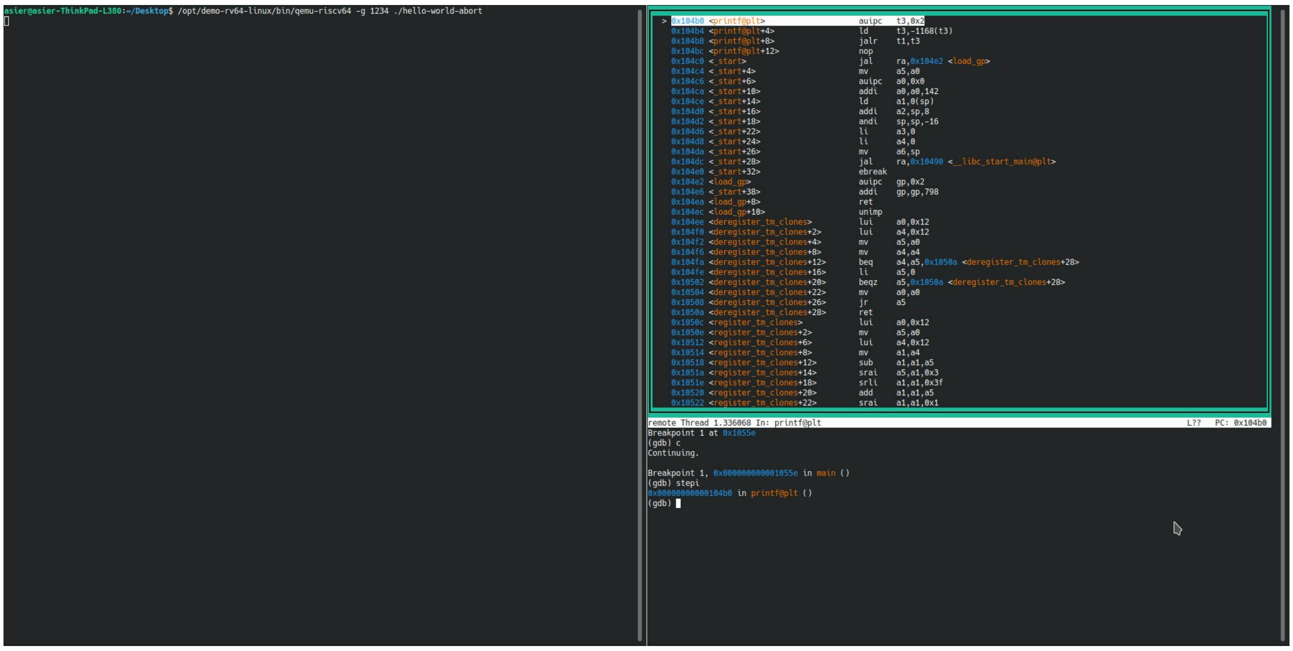
By entering the “c” command to continue, we can jump to the specified breakpoint in the main function. Notice how it stops at the call of the print function, as specified by the .c program.



```
asier@asier-ThinkPad-L380:~/Desktop$ /opt/demo-rv64-linux/bin/qemu-riscv64 -g 1234 ./hello-world-abort
```

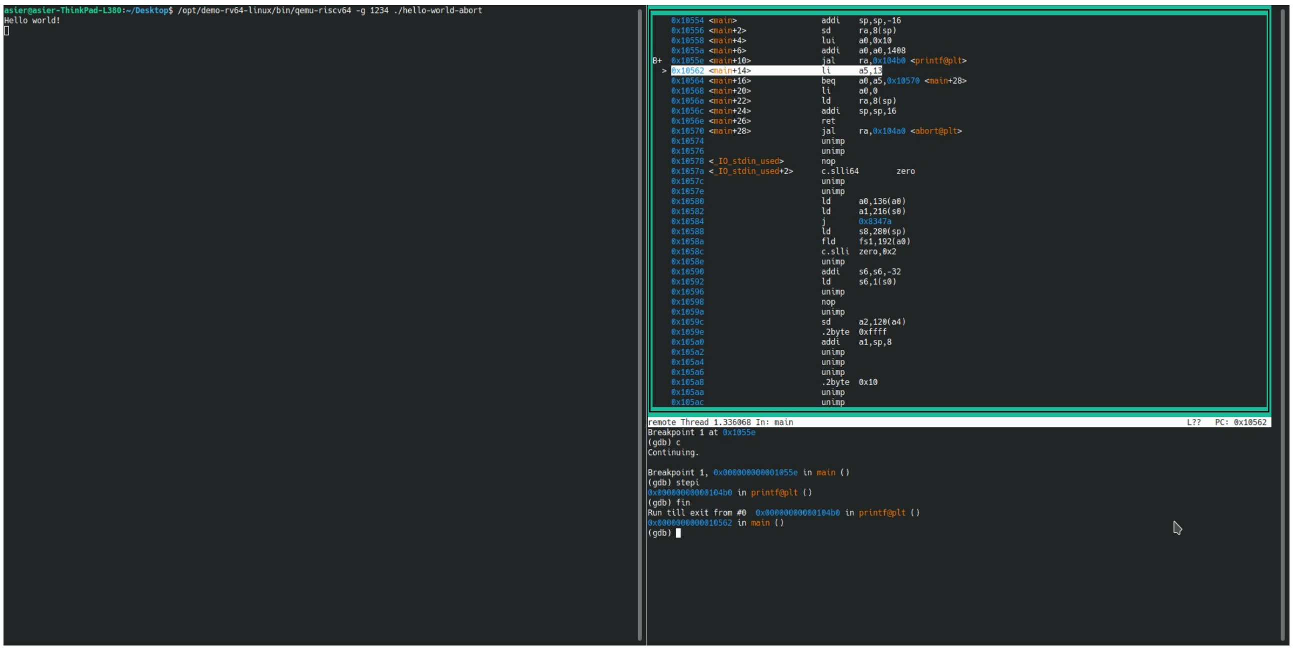
```
    0x10554 <main>                  addi    sp,sp,-16
    0x10556 <main+2>                sd      ra,8(sp)
    0x10558 <main+4>                lui     a0,0x10
    0x1055a <main+6>                addi    a0,a0,1408
B+> 0x1055e <main+10>               jal     ra,0x104b0 <printf@plt>
    0x10562 <main+14>               li      a5,13
    0x10564 <main+16>               beq     a0,a5,0x10570 <main+28>
    0x10568 <main+20>               li      a0,0
    0x1056a <main+22>               ld      ra,8(sp)
    0x1056c <main+24>               addi    sp,sp,16
    0x1056e <main+26>               ret
    0x10570 <main+28>               jal     ra,0x104a0 <abort@plt>
    0x10574                         unimp
    0x10576                         unimp
    0x10578 <_IO_stdin_used>        nop
    0x1057a <_IO_stdin_used+2>      c.slli64        zero
    0x1057c                         unimp
    0x1057e                         unimp
    0x10580                         ld      a0,136(a0)
    0x10582                         ld      a1,216(s0)
    0x10584                         j       0x8347a
    0x10588                         ld      s8,280(sp)
    0x1058a                         fld     fs1,192(a0)
    0x1058c                         c.slli  zero,0x2
    0x1058e                         unimp
    0x10590                         addi    s6,s6,-32
    0x10592                         ld      s6,1(s0)
    0x10596                         unimp
    0x10598                         nop
    0x1059a                         unimp
    0x1059c                         sd      a2,120(a4)
    0x1059e                         .2byte  0xffff
    0x105a0                         addi    a1,sp,8
    0x105a2                         unimp
    0x105a4                         unimp
    0x105a6                         unimp
    0x105a8                         .2byte  0x10
    0x105aa                         unimp
    0x105ac                         unimp
remote Thread 1.336068 In: main                                              L??   PC: 0x1055e
Breakpoint 1 at 0x1055e
(gdb) c
Continuing.

Breakpoint 1, 0x000000000001055e in main ()
(gdb) 
```

“stepi” can be used to enter the actual execution of the print function, and “fin” can be used to finish this step-in process. Notice how the “Hello world!” message appears on the left-most console, indicating that the execution of the program carries on as the debugger advances.



```
asier@asier-ThinkPad-L380:~/Desktop$ /opt/demo-rv64-linux/bin/qemu-riscv64 -g 1234 ./hello-world-abort
```

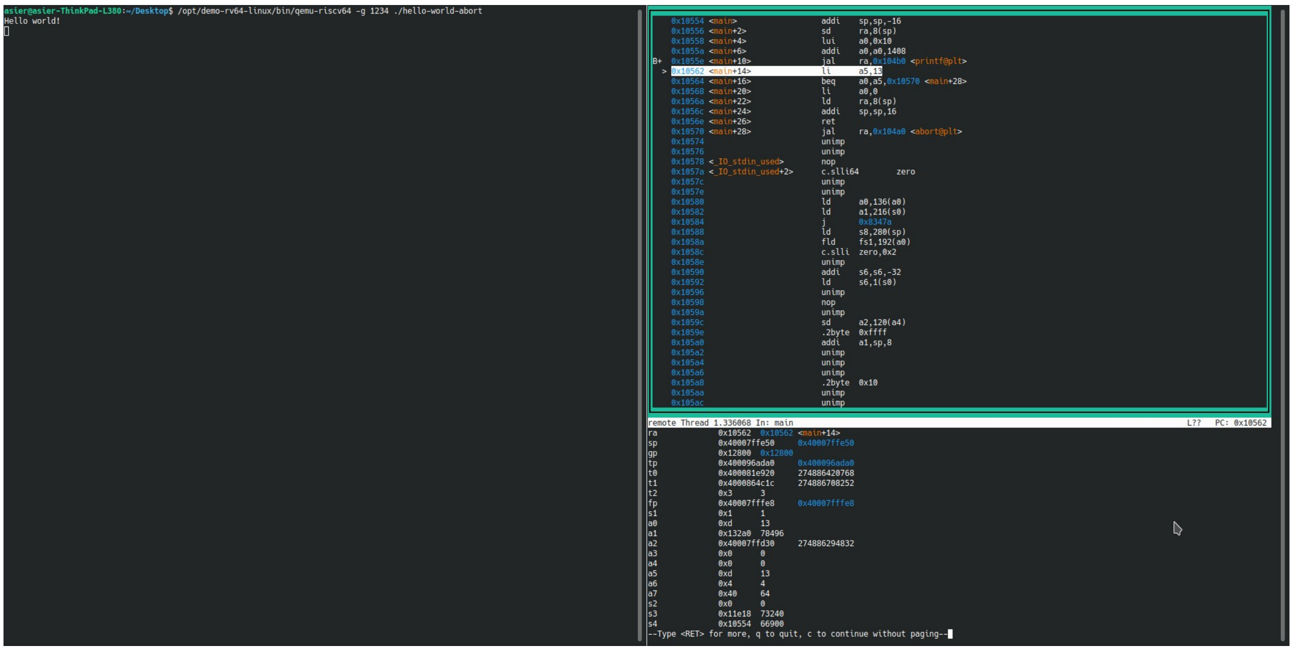
```
  > 0x104b0 <printf@plt>                    auipc   t3,0x2
    0x104b4 <printf@plt+4>                  ld      t3,-1168(t3)
    0x104b8 <printf@plt+8>                  jalr    t1,t3
    0x104bc <printf@plt+12>                 nop
    0x104c0 <_start>                        jal     ra,0x104e2 <load_gp>
    0x104c4 <_start+4>                      mv      a5,a0
    0x104c6 <_start+6>                      auipc   a0,0x0
    0x104ca <_start+10>                     addi    a0,a0,142
    0x104ce <_start+14>                     ld      a1,0(sp)
    0x104d0 <_start+16>                     addi    a2,sp,8
    0x104d2 <_start+18>                     andi    sp,sp,-16
    0x104d6 <_start+22>                     li      a3,0
    0x104d8 <_start+24>                     li      a4,0
    0x104da <_start+26>                     mv      a6,sp
    0x104dc <_start+28>                     jal     ra,0x10490 <__libc_start_main@plt>
    0x104e0 <_start+32>                     ebreak
    0x104e2 <load_gp>                       auipc   gp,0x2
    0x104e6 <_start+38>                     addi    gp,gp,798
    0x104ea <load_gp+8>                     ret
    0x104ec <load_gp+10>                    unimp
    0x104ee <deregister_tm_clones>          lui     a0,0x12
    0x104f0 <deregister_tm_clones+2>        lui     a4,0x12
    0x104f2 <deregister_tm_clones+4>        mv      a5,a0
    0x104f6 <deregister_tm_clones+8>        mv      a4,a4
    0x104fa <deregister_tm_clones+12>       beq     a4,a5,0x1050a <deregister_tm_clones+28>
    0x104fe <deregister_tm_clones+16>       li      a5,0
    0x10502 <deregister_tm_clones+20>       beqz    a5,0x1050a <deregister_tm_clones+28>
    0x10504 <deregister_tm_clones+22>       mv      a0,a0
    0x10508 <deregister_tm_clones+26>       jr      a5
    0x1050a <deregister_tm_clones+28>       ret
    0x1050c <register_tm_clones>            lui     a0,0x12
    0x1050e <register_tm_clones+2>          mv      a5,a0
    0x10512 <register_tm_clones+6>          lui     a4,0x12
    0x10514 <register_tm_clones+8>          mv      a1,a4
    0x10518 <register_tm_clones+12>         sub     a1,a1,a5
    0x1051a <register_tm_clones+14>         srai    a5,a1,0x3
    0x1051e <register_tm_clones+18>         srli    a1,a1,0x3f
    0x10520 <register_tm_clones+20>         add     a1,a1,a5
    0x10522 <register_tm_clones+22>         srai    a1,a1,0x1
remote Thread 1.336068 In: printf@plt                                            L??   PC: 0x104b0
Breakpoint 1 at 0x1055e
(gdb) c
Continuing.

Breakpoint 1, 0x000000000001055e in main ()
(gdb) stepi
0x00000000000104b0 in printf@plt ()
(gdb) 
```



```
asier@asier-ThinkPad-L380:~/Desktop$ /opt/demo-rv64-linux/bin/qemu-riscv64 -g 1234 ./hello-world-abort
Hello world!
```

```
    0x10554 <main>                  addi    sp,sp,-16
    0x10556 <main+2>                sd      ra,8(sp)
    0x10558 <main+4>                lui     a0,0x10
    0x1055a <main+6>                addi    a0,a0,1408
B+  0x1055e <main+10>               jal     ra,0x104b0 <printf@plt>
  > 0x10562 <main+14>               li      a5,13
    0x10564 <main+16>               beq     a0,a5,0x10570 <main+28>
    0x10568 <main+20>               li      a0,0
    0x1056a <main+22>               ld      ra,8(sp)
    0x1056c <main+24>               addi    sp,sp,16
    0x1056e <main+26>               ret
    0x10570 <main+28>               jal     ra,0x104a0 <abort@plt>
    0x10574                         unimp
    0x10576                         unimp
    0x10578 <_IO_stdin_used>        nop
    0x1057a <_IO_stdin_used+2>      c.slli64        zero
    0x1057c                         unimp
    0x1057e                         unimp
    0x10580                         ld      a0,136(a0)
    0x10582                         ld      a1,216(s0)
    0x10584                         j       0x8347a
    0x10588                         ld      s8,280(sp)
    0x1058a                         fld     fs1,192(a0)
    0x1058c                         c.slli  zero,0x2
    0x1058e                         unimp
    0x10590                         addi    s6,s6,-32
    0x10592                         ld      s6,1(s0)
    0x10596                         unimp
    0x10598                         nop
    0x1059a                         unimp
    0x1059c                         sd      a2,120(a4)
    0x1059e                         .2byte  0xffff
    0x105a0                         addi    a1,sp,8
    0x105a2                         unimp
    0x105a4                         unimp
    0x105a6                         unimp
    0x105a8                         .2byte  0x10
    0x105aa                         unimp
    0x105ac                         unimp
remote Thread 1.336068 In: main                                                  L??   PC: 0x10562
Breakpoint 1 at 0x1055e
(gdb) c
Continuing.

Breakpoint 1, 0x000000000001055e in main ()
(gdb) stepi
0x00000000000104b0 in printf@plt ()
(gdb) fin
Run till exit from #0  0x00000000000104b0 in printf@plt ()
0x0000000000010562 in main ()
(gdb) 
```

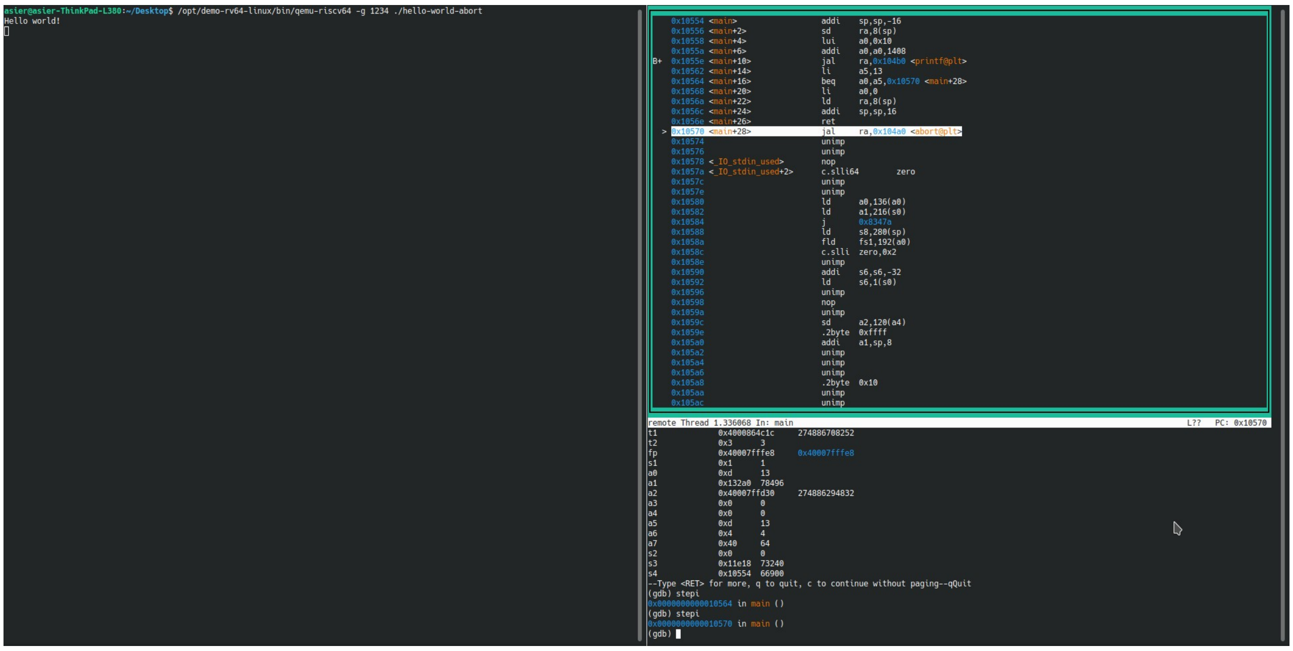
“i reg” can be used to inspect the values of the CPU registers, just as in any debugging interface.



```
asier@asier-ThinkPad-L380:~/Desktop$ /opt/demo-rv64-linux/bin/qemu-riscv64 -g 1234 ./hello-world-abort
Hello world!
```

```
    0x10554 <main>                  addi    sp,sp,-16
    0x10556 <main+2>                sd      ra,8(sp)
    0x10558 <main+4>                lui     a0,0x10
    0x1055a <main+6>                addi    a0,a0,1408
B+  0x1055e <main+10>               jal     ra,0x104b0 <printf@plt>
  > 0x10562 <main+14>               li      a5,13
    0x10564 <main+16>               beq     a0,a5,0x10570 <main+28>
    0x10568 <main+20>               li      a0,0
    0x1056a <main+22>               ld      ra,8(sp)
    0x1056c <main+24>               addi    sp,sp,16
    0x1056e <main+26>               ret
    0x10570 <main+28>               jal     ra,0x104a0 <abort@plt>
    0x10574                         unimp
    0x10576                         unimp
    0x10578 <_IO_stdin_used>        nop
    0x1057a <_IO_stdin_used+2>      c.slli64        zero
    0x1057c                         unimp
    0x1057e                         unimp
    0x10580                         ld      a0,136(a0)
    0x10582                         ld      a1,216(s0)
    0x10584                         j       0x8347a
    0x10588                         ld      s8,280(sp)
    0x1058a                         fld     fs1,192(a0)
    0x1058c                         c.slli  zero,0x2
    0x1058e                         unimp
    0x10590                         addi    s6,s6,-32
    0x10592                         ld      s6,1(s0)
    0x10596                         unimp
    0x10598                         nop
    0x1059a                         unimp
    0x1059c                         sd      a2,120(a4)
    0x1059e                         .2byte  0xffff
    0x105a0                         addi    a1,sp,8
    0x105a2                         unimp
    0x105a4                         unimp
    0x105a6                         unimp
    0x105a8                         .2byte  0x10
    0x105aa                         unimp
    0x105ac                         unimp
remote Thread 1.336068 In: main                                                      L??   PC: 0x10562
ra             0x10562  0x10562 <main+14>
sp             0x40007ffe50     0x40007ffe50
gp             0x12800  0x12800
tp             0x400096ada0     0x400096ada0
t0             0x400081e920     274886420768
t1             0x4000864c1c     274886708252
t2             0x3      3
fp             0x40007fffe8     0x40007fffe8
s1             0x1      1
a0             0xd      13
a1             0x132a0  78496
a2             0x40007ffd30     274886294832
a3             0x0      0
a4             0x0      0
a5             0xd      13
a6             0x4      4
a7             0x40     64
s2             0x0      0
s3             0x11e18  73240
s4             0x10554  66900
--Type <RET> for more, q to quit, c to continue without paging--
```

Finally, we can “stepi” our way into the abort function call, which stops the program’s execution.



```
asier@asier-ThinkPad-L380:~/Desktop$ /opt/demo-rv64-linux/bin/qemu-riscv64 -g 1234 ./hello-world-abort
Hello world!
```

```
    0x10554 <main>                  addi    sp,sp,-16
    0x10556 <main+2>                sd      ra,8(sp)
    0x10558 <main+4>                lui     a0,0x10
    0x1055a <main+6>                addi    a0,a0,1408
B+  0x1055e <main+10>               jal     ra,0x104b0 <printf@plt>
    0x10562 <main+14>               li      a5,13
    0x10564 <main+16>               beq     a0,a5,0x10570 <main+28>
    0x10568 <main+20>               li      a0,0
    0x1056a <main+22>               ld      ra,8(sp)
    0x1056c <main+24>               addi    sp,sp,16
    0x1056e <main+26>               ret
  > 0x10570 <main+28>               jal     ra,0x104a0 <abort@plt>
    0x10574                         unimp
    0x10576                         unimp
    0x10578 <_IO_stdin_used>        nop
    0x1057a <_IO_stdin_used+2>      c.slli64        zero
    0x1057c                         unimp
    0x1057e                         unimp
    0x10580                         ld      a0,136(a0)
    0x10582                         ld      a1,216(s0)
    0x10584                         j       0x8347a
    0x10588                         ld      s8,280(sp)
    0x1058a                         fld     fs1,192(a0)
    0x1058c                         c.slli  zero,0x2
    0x1058e                         unimp
    0x10590                         addi    s6,s6,-32
    0x10592                         ld      s6,1(s0)
    0x10596                         unimp
    0x10598                         nop
    0x1059a                         unimp
    0x1059c                         sd      a2,120(a4)
    0x1059e                         .2byte  0xffff
    0x105a0                         addi    a1,sp,8
    0x105a2                         unimp
    0x105a4                         unimp
    0x105a6                         unimp
    0x105a8                         .2byte  0x10
    0x105aa                         unimp
    0x105ac                         unimp
remote Thread 1.336068 In: main                                                      L??   PC: 0x10570
t1             0x4000864c1c     274886708252
t2             0x3      3
fp             0x40007fffe8     0x40007fffe8
s1             0x1      1
a0             0xd      13
a1             0x132a0  78496
a2             0x40007ffd30     274886294832
a3             0x0      0
a4             0x0      0
a5             0xd      13
a6             0x4      4
a7             0x40     64
s2             0x0      0
s3             0x11e18  73240
s4             0x10554  66900
--Type <RET> for more, q to quit, c to continue without paging--qQuit
(gdb) stepi
0x0000000000010564 in main ()
(gdb) stepi
0x0000000000010570 in main ()
(gdb) 
```