



Professional development tools for RISC-V

Ryan Sheng, ryan.sheng@iar.com, 021-63758658
IAR Systems (China)
2019.12.20

IAR Systems

- World-leading embedded development tools vendor
- Established in 1983
- Headquartered in Uppsala, Sweden
- 200 employees, 11 offices in EMEA / APAC / US
- Listed on Stockholm/NASDAQ-OMX

**13,000+
SUPPORTED
DEVICES**

**150,000
USERS
WORLDWIDE**

**TÜV
SÜD**
Safety tested Production monitored Functional Safety

Complete Arm 32-bit support

Complete Renesas MCU support

Renesas ABI compliant

**IAR Embedded Workbench
for RISC-V**

RISC-V®

**Fully integrated runtime
and static analysis tools**

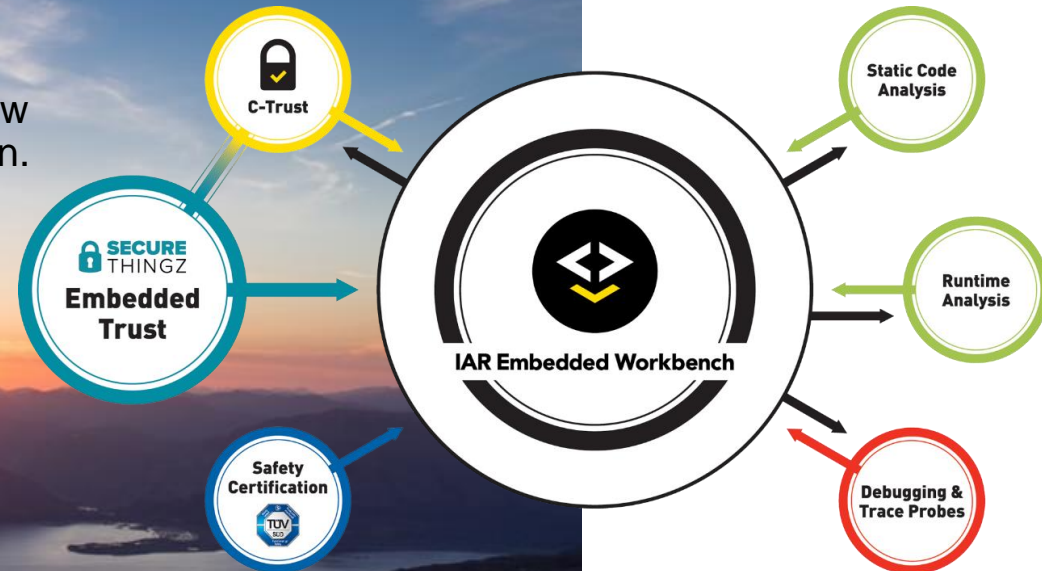
C-RUN C-STAT

CMSIS COMPLIANT
ARM® Cortex® Microcontroller
Software Interface Standard

Total quality • Total safety • Total security

Total tools

Our customers build the technology for a new world. We supply the tools to make it happen. One toolbox, one view, one uninterrupted workflow. As simple as that.



IAR Embedded Workbench C/C++ Compiler and Debugger IDE

Most widely used embedded software development tools
User-friendly IDE features and broad ecosystem integration
Industry leading optimization for code size and speed

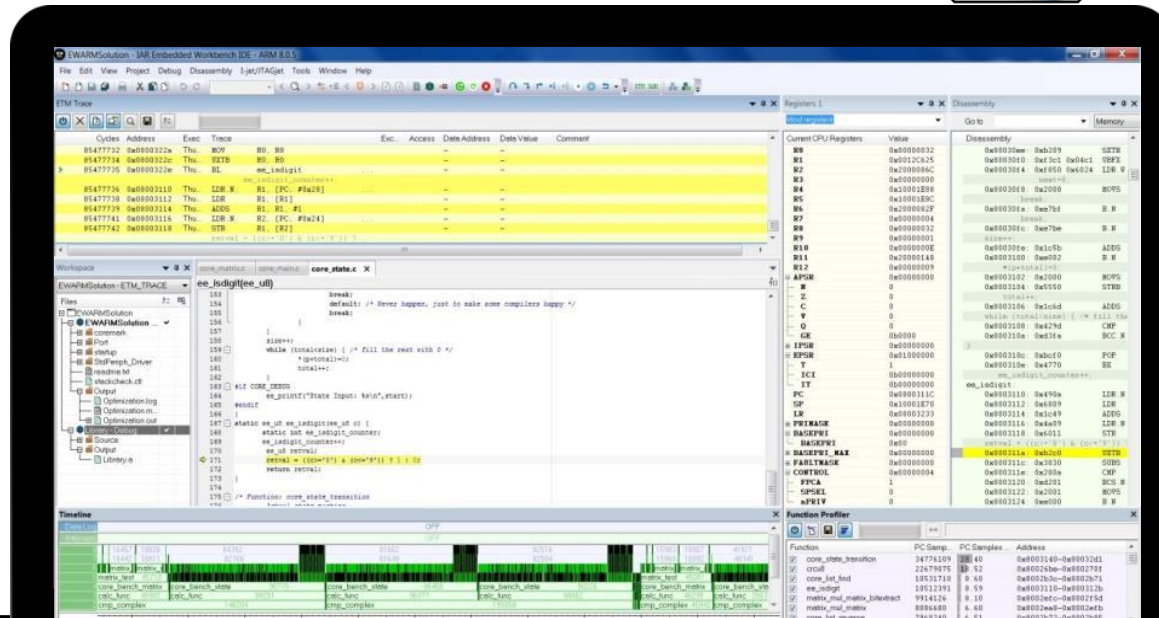
ISO/ANSI compliance
with C18 and C++17

Comprehensive graphical
debugger interface

Integrated code analysis
add-ons

Functional safety certified

Global support & service



Support for 13,000+ devices

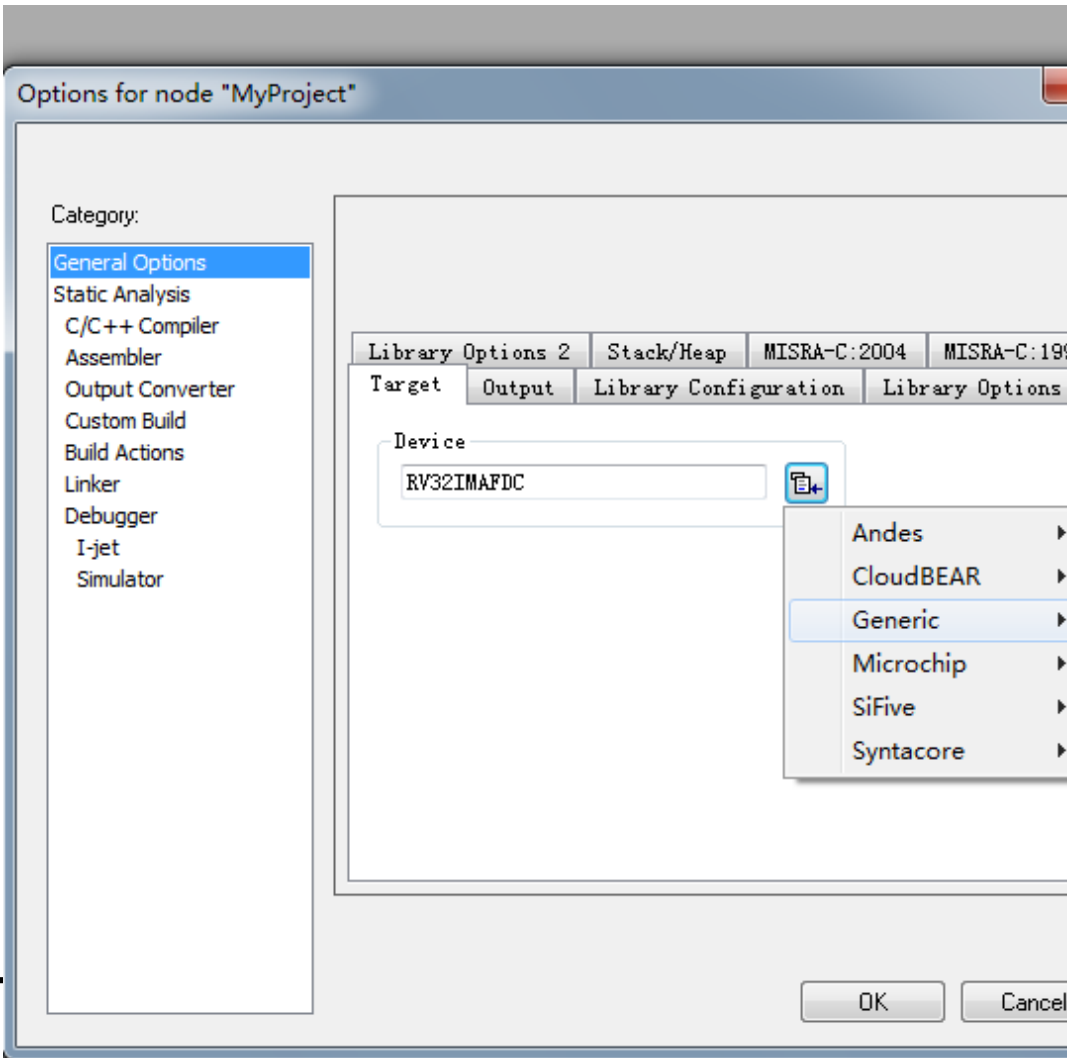
Different architecture, One solution

All available 8-, 16- and 32-bit MCUs

Cortex-M0	Cortex-R8	AVR	H8
Cortex-M0+	Cortex-A5	AVR32	STM8
Cortex-M1	Cortex-A7	RX	ColdFire
Cortex-M3	Cortex-A8	RL78	HCS12
Cortex-M4	Cortex-A9	RH850	S08
Cortex-M7	Cortex-A15	78K	MAXQ
Cortex-M23	ARM11	SuperH	CR16C
Cortex-M33	ARM9	V850	SAM8
Cortex-R4	ARM7	R32C	RISC-V
Cortex-R5	SecurCore	M32C	
Cortex-R52	8051	M16C	
Cortex-R7	MSP430	R8C	



Device support for RISC-V



RV32EM
 RV32EMA
 RV32EMAC
 RV32EMAF
 RV32EMAF C
 RV32EMAFD
 RV32EMAFDC
 RV32EMC
 RV32EMF
 RV32EMFC
 RV32EMFD
 RV32EMFDC
 RV32IM
 RV32IMA
 RV32IMAC
 RV32IMAF
 RV32IMAF C
 RV32IMAFD
 RV32IMAFDC
 RV32IMC
 RV32IMF
 RV32IMFC
 RV32IMFD
 RV32IMFDC

RV32I

Base Integer Instruction Set

RV32E

Base Integer Instruction Set
 (embedded, 16 registers)

Supported extensions

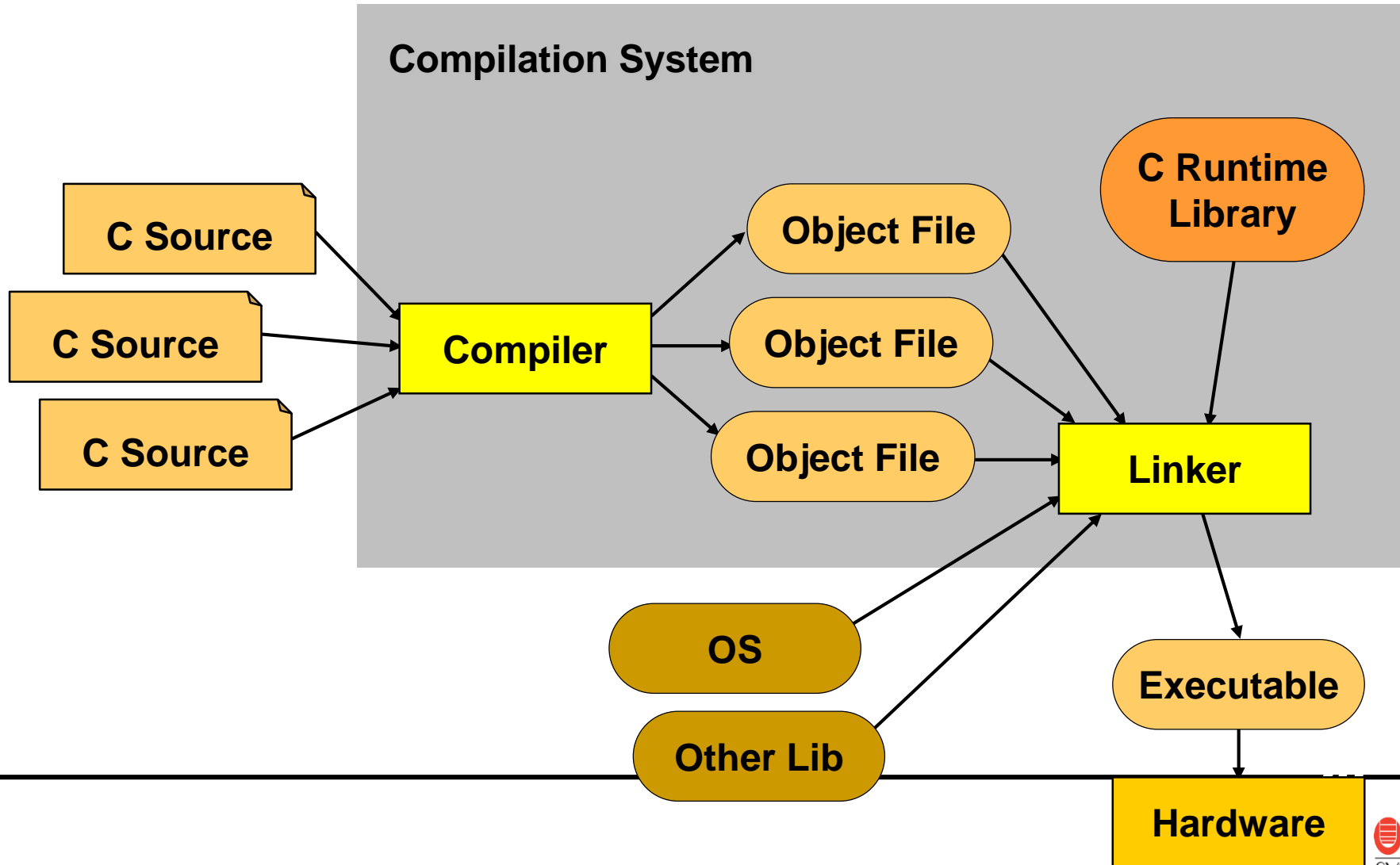
M	integer mul & div
A	atomic
F	single precision float
D	double precision float
C	compressed

Supported IP vendors

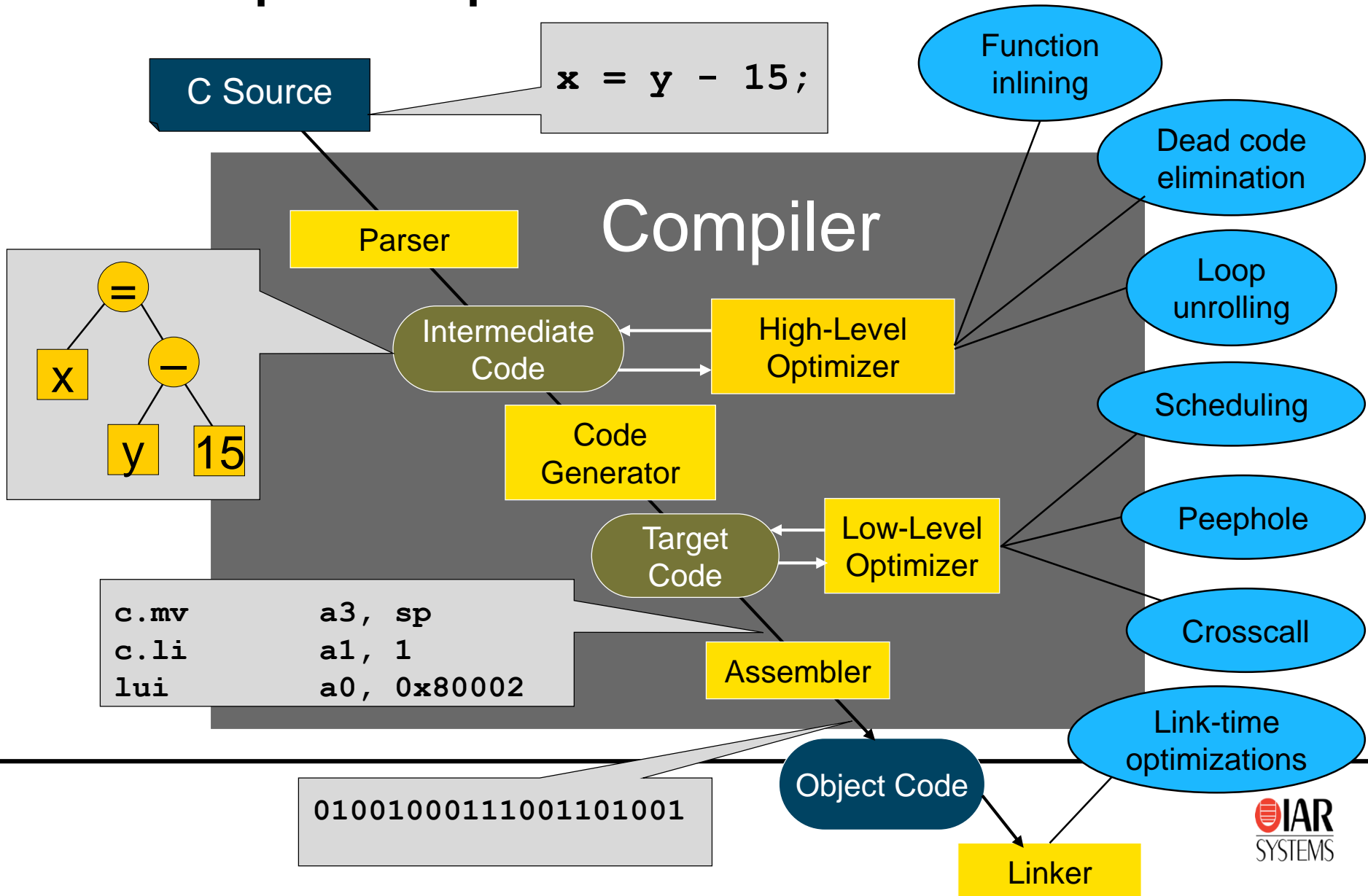
Andes
 CloudBEAR
 Microchip
 SiFive
 Syntacore

.....

Compilation system



Compiler optimizations



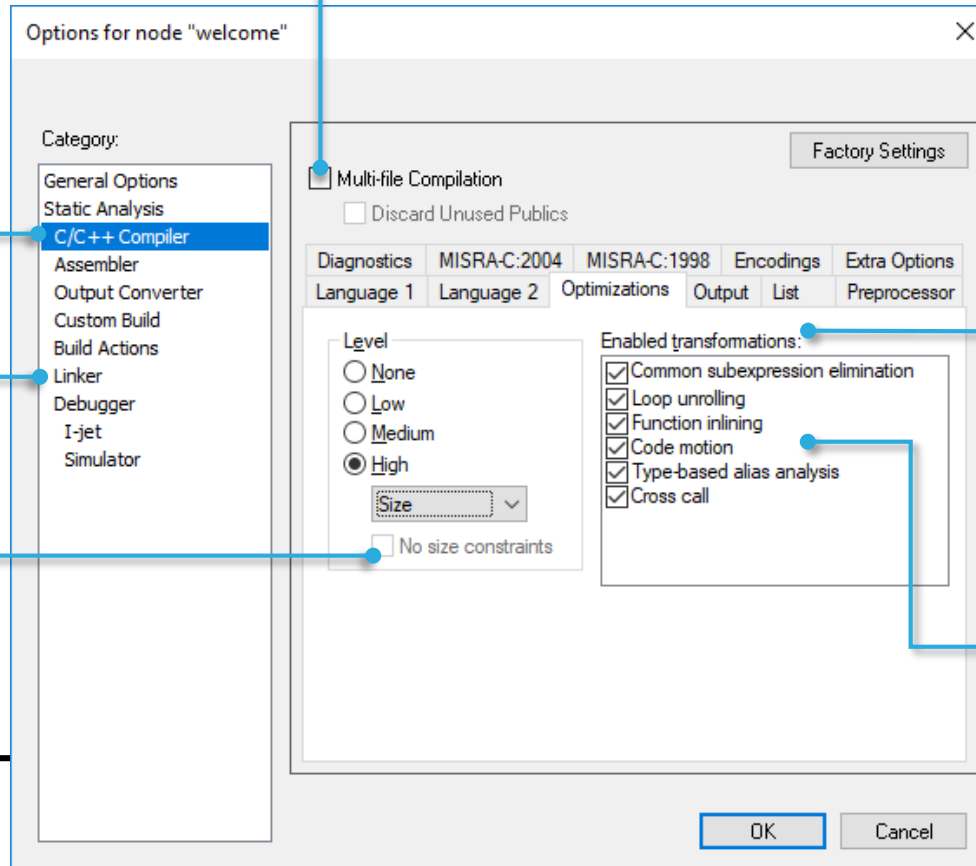
Controlling optimizations

Multiple optimization levels for code size and execution speed

The linker can remove unused code

Option to maximize speed with no size constraints

Multi-file compilation allows the optimizer to operate on a larger set of code



Language standards

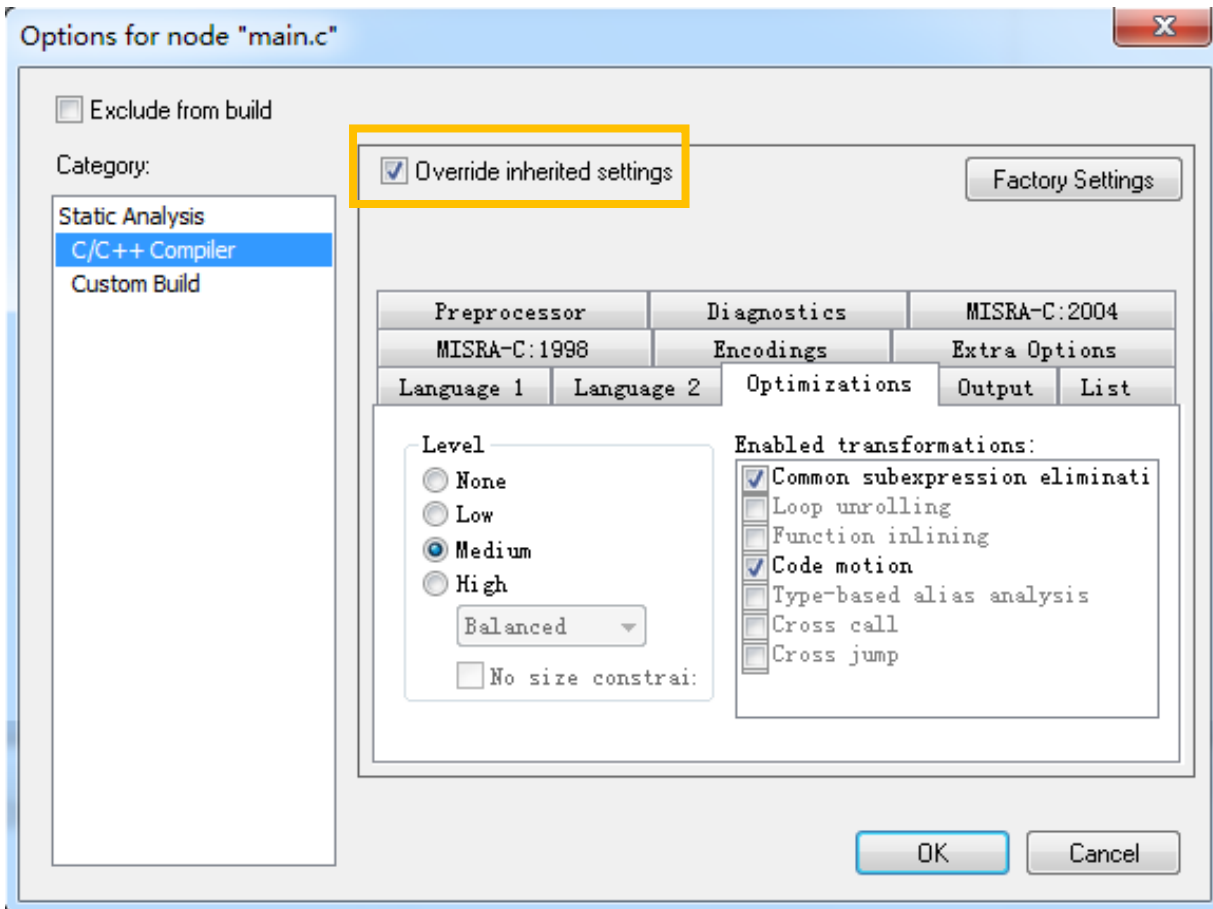
- ISO/IEC 14882:2015 (C++14, C++17)
- ISO/IEC 9899:2018 (C18)
- ANSI X3.159-1989 (C89)

- IEEE 754 standard for floating-point arithmetic

Major features of the optimizer can be controlled individually

Balance between size and speed by setting different optimizations for different parts of the code

Controlling optimizations



```
#pragma optimize=high
unsigned int GetFib(int n)
{
    if ((n > 0) && (n <= MAX_FIB))
    {
        return (Fib[n-1]);
    }
    else
    {
        return 0;
    }
}
```

Speed, size or both?

Optimization	Effect	
Common sub-expressions	Speed ↑	Size ↓
Loop unrolling	Speed ↑	Size ↑
Function inlining	Speed ↑	Size ↑
Code motion	Speed ↑	Size →
Dead code elimination	Speed →	Size ↓
Static clustering	Speed ↑	Size ↓
Instruction scheduling	Speed ↑	Size →
Peephole	Speed ↑	Size ↓
Cross call	Speed ↓	Size ↓

Challenges on optimization

- **Size**

- Compared to more complex instruction sets, RISC-V have challenges especially when it comes to code size
- Arithmetic with higher resolution than the natural data size yields larger code
- Absence of carry flags and instructions to save and restore multiple registers

- **Speed**

- When it comes to speed, RISC-V is competitive
- More speed optimizations in future releases

Our initial target will be reducing code size for small embedded systems. Our main focus has always been to supply the best code size and speed on the market.

GCC attributes

- In the extended language mode, IAR C/C++ compiler supports a selection of commonly used GCC-style attributes
- Use the `__attribute__((attribute-list))` syntax for these attributes
- The following attributes are supported in part or in whole

alias

aligned

always_inline

constructor

deprecated

noinline

noreturn

packed

pcs

section

target

transparent_
union

unused

used

volatile

weak

Custom instructions

- The `.insn` directive generates custom instructions which are not directly supported by the assembler
- The `.insn` directive generates instructions on all RISC-V instruction formats

<code>.insn</code> directives		
<code>.insn r</code>	<code>op7, f3, f7, rd, rs1, rs2</code>	
<code>.insn r</code>	<code>op7, f3, f7, rd, rs1, rs2, rs3</code>	
<code>.insn r4</code>	<code>op7, f3, f2, rd, rs1, rs2, rs3</code>	
<code>.insn i</code>	<code>op7, f3, rd, rs1, expr</code>	
<code>.insn i</code>	<code>op7, f3, rd, rs1, expr (rs1)</code>	
<code>.insn s</code>	<code>op7, f3, rd, rs1, expr (rs1)</code>	op2, op7 unsigned immediate 2 or 7-bit opcode
<code>.insn sb</code>	<code>op7, f3, rd, rs1, expr</code>	
<code>.insn sb</code>	<code>op7, f3, rd, expr(rs1)</code>	fn unsigned immediate for function code 2-7 bits wide
<code>.insn b</code>	<code>op7, f3, rd, rs1, expr</code>	
<code>.insn u</code>	<code>op7, f3, rd, expr</code>	
<code>.insn uj</code>	<code>op2, rd, expr</code>	rd, rsN register field integer (x0-x31) or FP (f0-f31)
<code>.insn cr</code>	<code>op2, f4, rd, rs1</code>	
<code>.insn ci</code>	<code>op2, f2, rd, expr</code>	
<code>.insn ciw</code>	<code>op2, f3, rd', expr</code>	rd', rsN' compact instruction register field integer (x8-x15) or FP (f8-f15)
<code>.insn ca</code>	<code>op2, f6, f2, rd', rs2'</code>	
<code>.insn cb</code>	<code>op2, f3, rs1', expr</code>	
<code>.insn cj</code>	<code>op2, f3, expr</code>	
<code>.insn cs</code>	<code>op2, f3, rs1', rs2', expr</code>	expr immediate expression

* Please refer to the RISC-V ISA specification sections 2.3 and 12.2 for details on bit-layout

Custom instructions: Example

- The `.insn` directive can be used to inline assembly code in programs written in C and C++
- Built-in constants are available when generating a custom instruction

Intrinsic-like `function` example

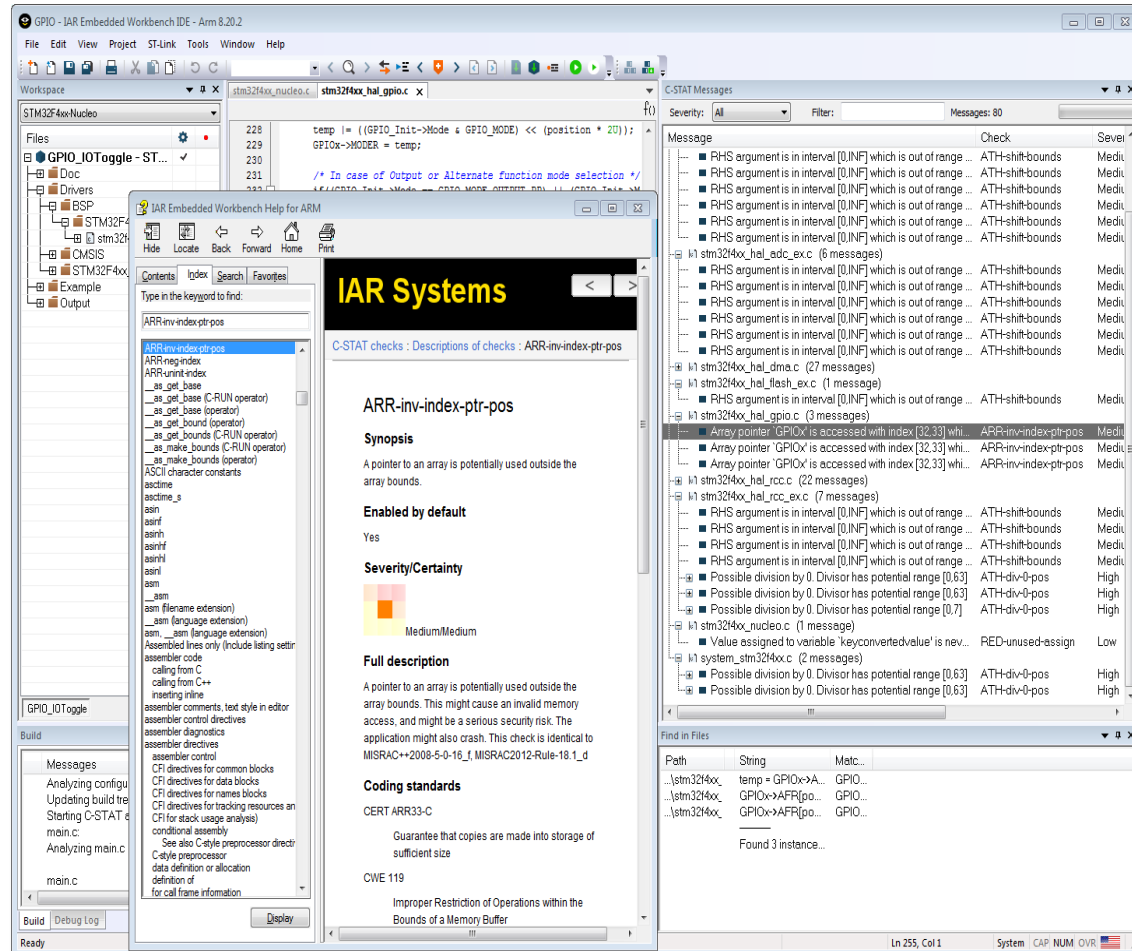
```
long __insn_example(int lhs, int rhs) {
    long res;
    /* Generates AND r,r,r */
    asm (".insn r 0x33, 0x7, 0x0, %0, %1, %2" \
        : "=r" (res) \
        : "r" (lhs), "r" (rhs) );
    return res;
}
```

Intrinsic-like `macro` example

```
/* Generates AND r,r,r */
#define __insn_example(lhs, rhs) ({ \
    int __lhs = (lhs), __rhs = (rhs), __res; \
    asm (".insn r 0x33, 0x7, 0x0, %0, %1, %2" \
        : "=r" (__res) \
        : "r" (__lhs), "r" (__rhs)); \
    __res; \
})
```

C-STAT: static code analysis

- Advanced C/C++ code analysis
- Fully integrated within IAR Embedded Workbench
- Check the compliance with **MISRA C:2004**, **MISRA C++:2008** and **MISRA C:2012**
- 250+ checks mapping to hundreds of issues covered by **CWE** and **CERT C/C++**
- Intuitive and easy-to-use settings
- Flexible rules selection
- Extensive and detailed documentation



CWE (Common Weakness Enumeration):
CERT (Computer Emergency Response Team):

<http://cwe.mitre.org>
<http://www.cert.org>

RISC-V debugging

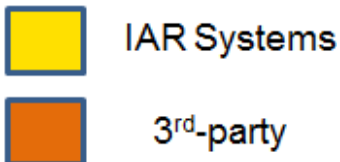
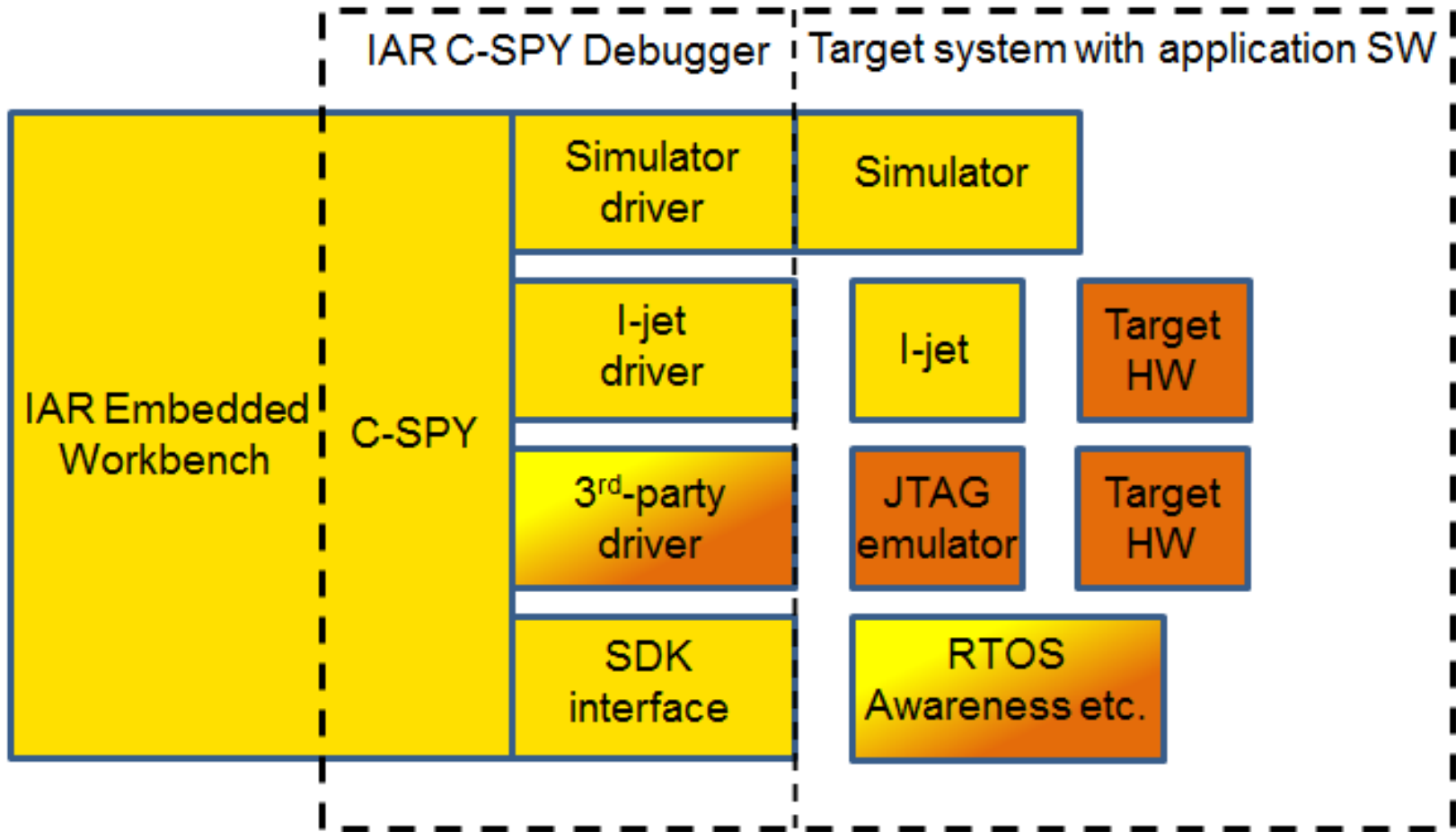
- IAR supports the latest complete RISC-V debug spec, currently v0.13
 - Any additional updates will continuously be supported
- Automated discovery of implemented debug features in a MCU or SoC
 - Implemented debug features like h/w breakpoints, supported extensions etc. are automatically read on connection
- Interrupt and exception catching
 - Distinguish between different priority levels and exception types
- Set different types of breakpoints
 - Code, data, log, trace start and stop, etc.
- Single step on both C/C++ and assembler level
- Full low-level access to all registers, memories and other resources on the MCU or SoC
- Script/macro execution capabilities

Debug & Trace probes

	I-jet	I-jet Trace (4-bit model)	I-jet Trace(16-bit model)
JTAG/SWD speed	48 MHz	100 MHz	100 MHz
Download speed (RAM)	1.89 MByte/s	3.73 MByte/s	3.73 MByte/s
SWO max. bandwidth	~30 Mbps	~60 Mbps	~60 Mbps
Available trace memory	-	64M or 256M bytes	256M or 1G bytes
Trace max. bandwidth	-	1.2 Gbps	11.2 Gbps
Max streaming speed	48 MByte/s	~380 MByte/s	~380 MByte/s
Power sampling resolution	~160 μ A	~160 μ A	~160 μ A
Power sampling rate	200 ksps	200 ksps	200 ksps



C-SPY debugger overview

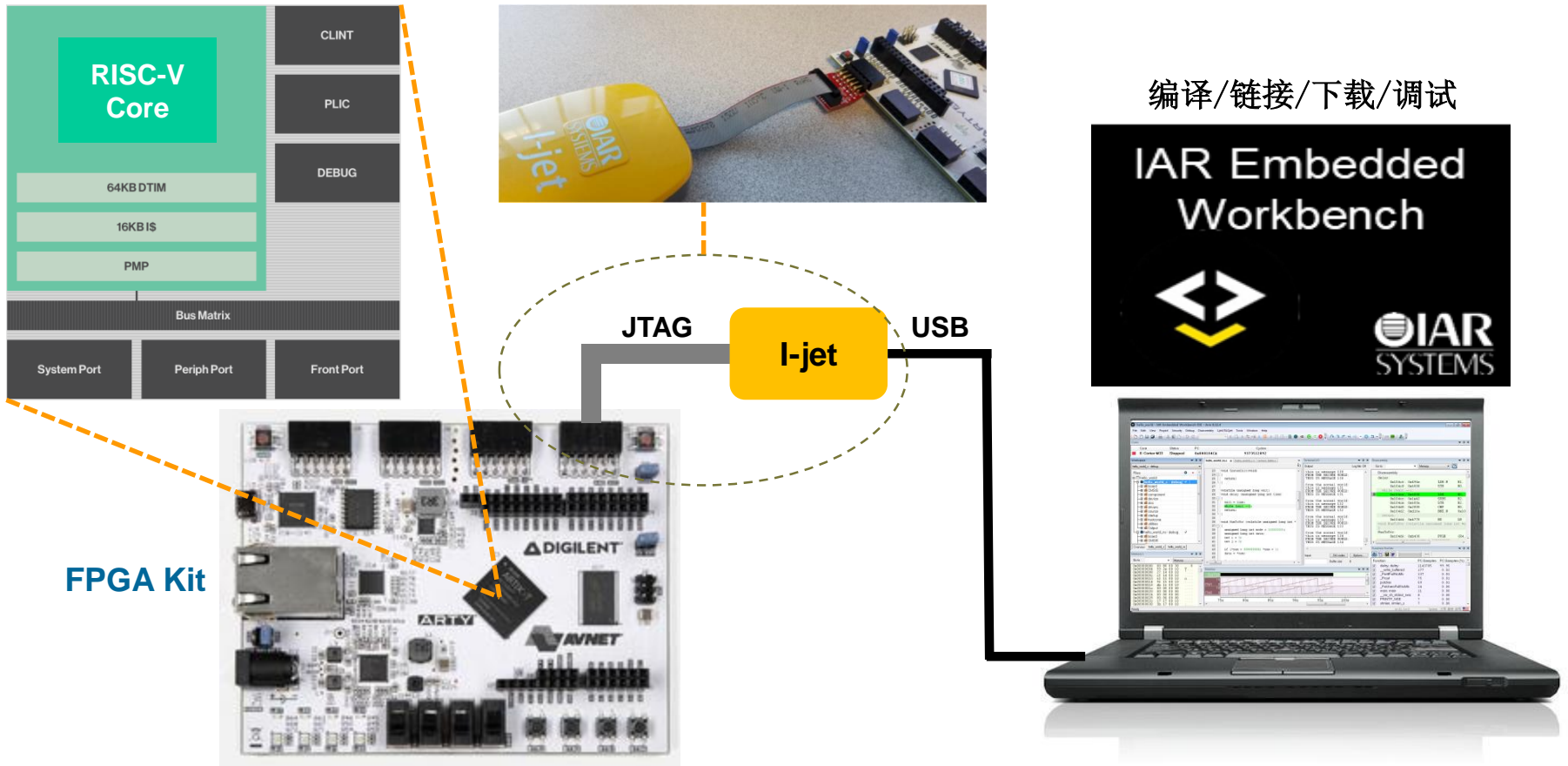


C-SPY debugger implementation

The screenshot displays the IAR Embedded Workbench IDE interface for RISC-V 1.11.1, showing the C-SPY debugger implementation. The interface is divided into several panels:

- Project & File:** Shows the project structure for 'welcome - Debug', including files like 'bsp', 'fe300prci_d...', 'init.c', 'plic_driver.c', 'cstat', 'c_stat.c', 'src', 'corelexip...', 'Output', 'welcome.map', and 'welcome.out'.
- Registers:** Lists registers with their values and access permissions. For example, 'PWM0CFG' has value '0x00001400' and 'PWM0CMP0' has value '0x000000FE'.
- Semihosted Terminal I/O:** Shows the output of the program: 'Welcome to the E31 Coreplex IP FPGA Evaluation Kit!'.
- Source & disassembly level debugging:** Displays the C source code for 'main()' and its corresponding assembly instructions. The source code includes comments like '// The LEDs are intentionally left somewhat dim.' and code for setting PWM registers. The assembly shows instructions like 'c.addi a0, -1', 'c.addi a2, 1', and 'slli a4, a2, 0x10'.
- Stack usage:** Shows the stack frame for the 'main' function, with variables 'i', 'g', and 'b' at memory addresses 0x80000FE0, 0x80000FE8, and 0x80000FF0 respectively.
- Expressions monitoring:** Monitors expressions like 'r', 'g', and 'b' with their current values and locations. For example, 'r' has value 50 at location a1[0:15].
- Code & data breakpoints:** Lists breakpoints set in the source code, such as 'Code coreplexip_welcome.c:124.3' and 'Code coreplexip_welcome.c:144.5 when "r % 50 == 0 ...'.
- Variables monitoring:** Monitors local variables like 'i', 'r', 'g', 'b', 'now', and 'then' with their values and types. For example, 'now' has value '0x2000BFF8' and type 'uint64_t volatile *'.

Full development environment



Trace on RISC-V (coming ...)

- What is Trace ?

- In contrast to traditional debugging, trace is non-intrusively observing your application
- Capture the full PC flow
- Go back in time and see how you arrived at the current point
- Quickly isolate exceptions and hard faults
- Find bugs that are rare and dependent on the order-of-execution
- Performance and coverage monitoring, e.g. find where your application is spending its time, isolate the dead code, show test deficiencies, etc.

- RISC-V trace specifics

- Specifications of standard RISC-V trace are still under development
- Processor Trace TG defines trace encoder packets and the core→encoder interface
- More work is needed to make all aspects of trace standard (e.g. control & export)
- Goal is to get on par with what is already existing on more mature architectures

Trace on RISC-V (coming ...)

Instruction Trace

Timeline

Code Coverage

Instruction Coverage

The screenshot displays a RISC-V debugger interface with four main panels:

- Instruction Trace:** Shows a list of instructions with their timestamps and addresses. The instruction at address 40400852 is highlighted in yellow.
- Timeline:** A Gantt chart showing the execution of instructions over time. The instruction at address 40400852 is highlighted in yellow.
- Code Coverage:** A table showing the coverage percentage for various code blocks. The instruction at address 40400852 is highlighted in yellow.
- Disassembly:** Shows the assembly code for the instruction at address 40400852, which is highlighted in yellow.

Timestamp	Trace
16464	4040082C c.li a0, 1
16465	4040082E c.li a1, 1
16466	40400830 bge s0, a1, 0x4040084E
16467	4040084E andi a0, s0, 1
16468	40400852 c.beqz a0, 0x4040085E
16469	40400854 addi a0, s0, -1
16470	40400858 c.jal a_pow2

Code	Coverage (%)	Code Range
welcome(Program)	41.5	
IAR_Trace_Test(Module)	56.7	
a2_pow2	100.0	
a_pow2	88.9	
itt_get_mode	0.0	
itt_init	0.0	
itt_loop	100.0	
itt_run	39.3	
itt_set_cpu_freq	0.0	
itt_set_irq_freq	0.0	
itt_test0	0.0	
itt_test1	0.0	
itt_test2	100.0	
t2_pow2	77.8	
if (n <= 0)		0x4040094C-0x40400951
return 1 + ZERO_1;		0x40400952-0x4040096B
if (n & 0x1)		0x4040096C-0x40400971
v1 = a_pow2(n - 1);		0x40400972-0x4040097B
v1 = t2_pow2(n - 1);		0x4040097C-0x40400983
if (n & 0x2)		0x40400984-0x40400989
v2 = t_pow2(n - 1);		0x4040098A-0x40400991
v2 = a2_pow2(n - 1);		0x40400992-0x40400997
return v1 + v2 + Z...		0x40400998-0x404009BB
t_pow2	90.0	
xpow2	100.0	
__dbg_abort(Module)	0.0	
__dbg_break(Module)	0.0	
__dbg_xxexit(Module)	0.0	
coreplexip_welcome(Mo...	0.0	

C Disassembly
40400816 40B2 c.lwsp ra, 0xC
40400818 4422 c.lwsp s0, 8
4040081A 4492 c.lwsp s1, 4
4040081C 4902 c.lwsp s2, 0
4040081E 6141 c.addi16sp 0x10
40400820 8082 c.ret
__thumb unsigned int t_pow2(int n)
{
t_pow2:
40400822 717D c.addi16sp -0x10
40400824 C606 c.swsp ra, 0xC
40400826 C422 c.swsp s0, 8
40400828 C226 c.swsp s1, 4
4040082A 842A c.mv s0, a0
unsigned int v1 = 1 // Different code to make 't2_pow'
4040082C 4505 c.li a0, 1
if (n <= 0)
4040082E 4585 c.li a1, 1
40400830 00B45F63 bge s0, a1, 0x4040084E
return v1 + ZERO_1; // Complex of simple (to a
40400834 0181A583 lw a1, 0x18(gp)
40400838 01C1A603 lw a2, 0x1C(gp)
4040083C 95B2 c.add a1, a2
4040083E 0201A603 lw a2, 0x20(gp)
40400842 95B2 c.add a1, a2
40400844 0241A603 lw a2, 0x24(gp)
40400848 95B2 c.add a1, a2
4040084A 952E c.add a0, a1
4040084C A0A1 c.j 0x40400894
if (n & 0x1)
4040084E 00147513 andi a0, s0, 1
40400852 C511 c.beqz a0, 0x4040085E
v1 = a_pow2(n - 1);

Summary

- Meet your demand of quality & time-to-market
 - Easy code reuse and widest customers base from IAR Embedded Workbench
 - Fit the needs of both memory size and necessary performance by the outstanding C/C++ compiler
 - Improve the code quality and find potential issues earlier by the integrated C-STAT analysis
 - Identify low level bugs and provide graphical visibility to all SoC resource by the powerful debugger

Thanks for your attention !

www.iar.com/riscv