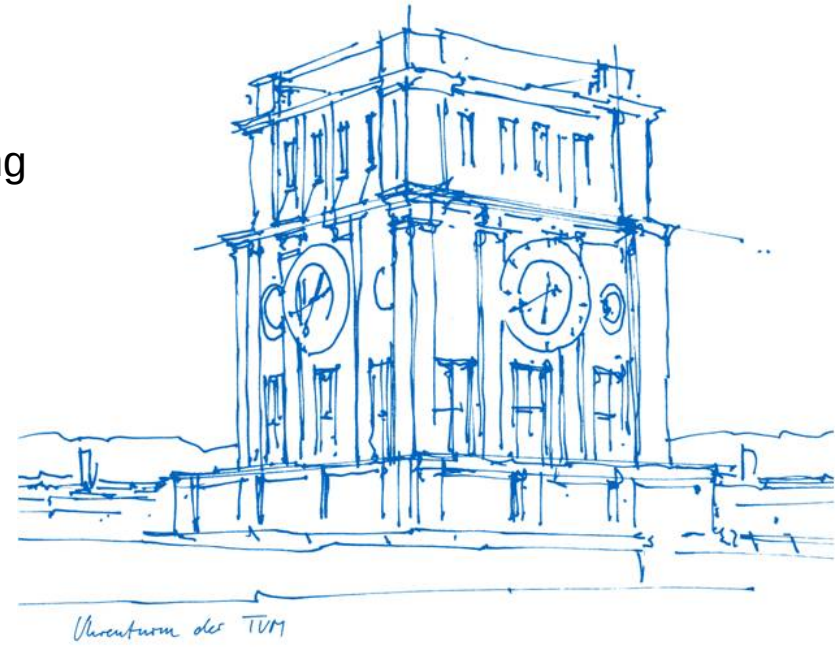
SeRoHAL:
Generation of Selectively Robust Hardware Abstraction Layers for Efficient Protection of Mixed-criticality Systems

Petra R. Kleeberger, Juana Rivera, Daniel Mueller-Gritschneder, Ulf Schlichtmann

Technical University of Munich

Department of Electrical and Computer Engineering

Chair of Electronic Design Automation



Agenda

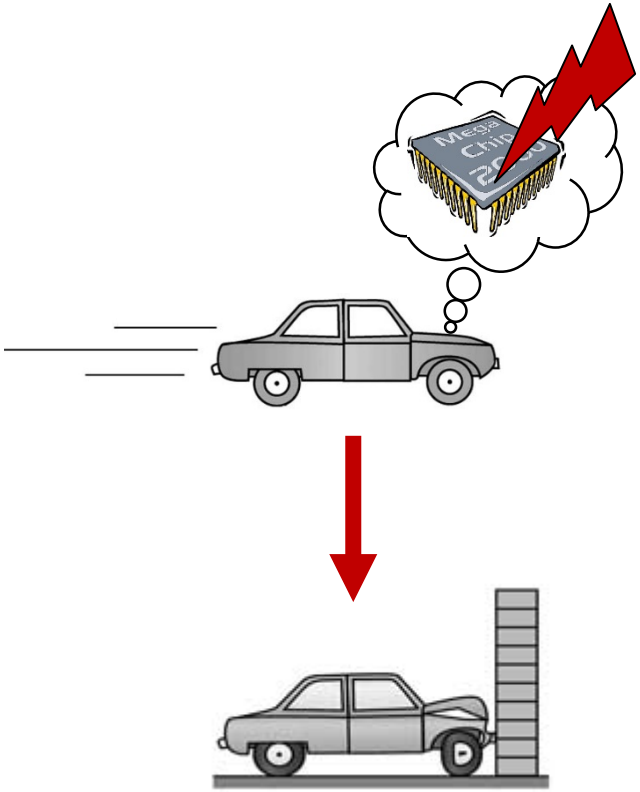
1. Introduction
2. SeRoHAL
   a) Template-based Generation of Robust Register-specific HAL
   b) Selective Protection for Mixed-criticality Systems
3. Results
4. Conclusion

Motivation

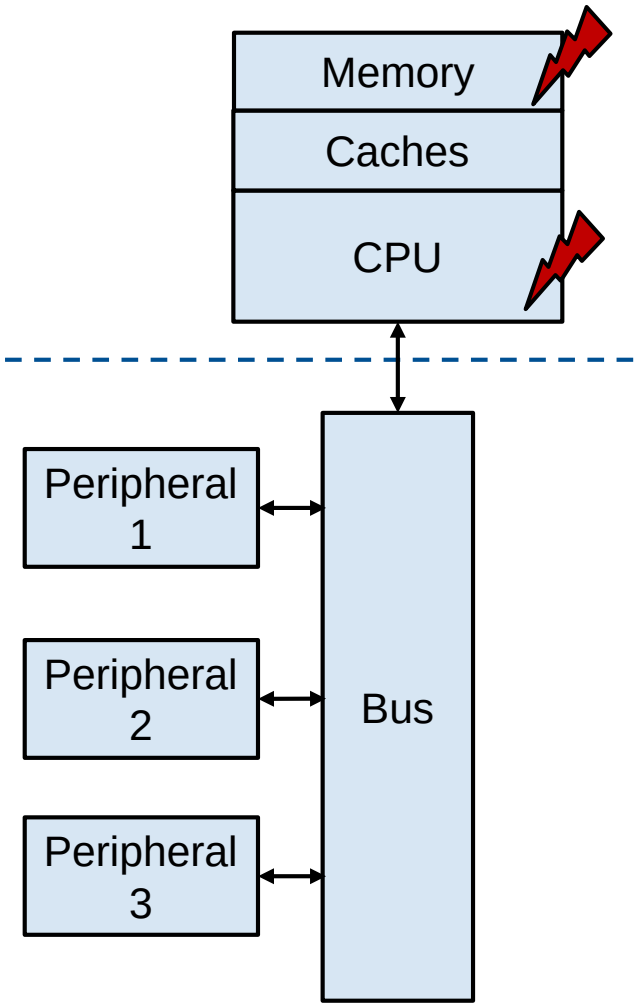
Hardware increasingly vulnerable to errors
- e.g. soft errors induced by radiation

Safety-critical tasks:
- Autonomous driving
- Brake-by-wire

➢ Safe operation in presence of errors must be ensured
➢ Error detection and handling required

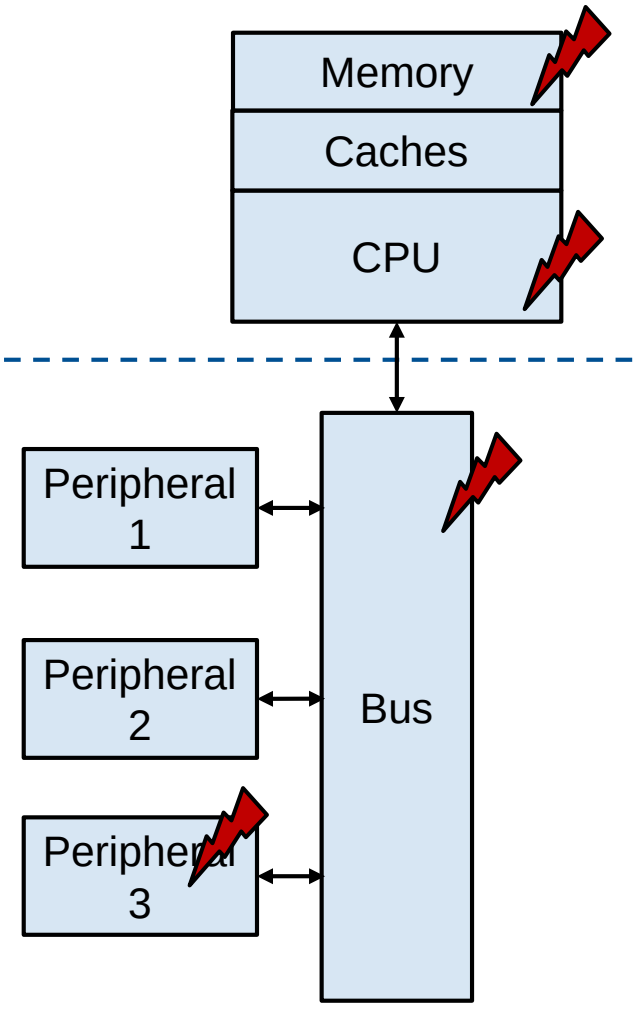


Error Detection and Handling



HW-based mechanisms:	SW-based mechanisms inserted by compilers:
• Lockstep • ECC for memory	• Instruction duplication • Variable duplication • Control-flow checking

Error Detection and Handling



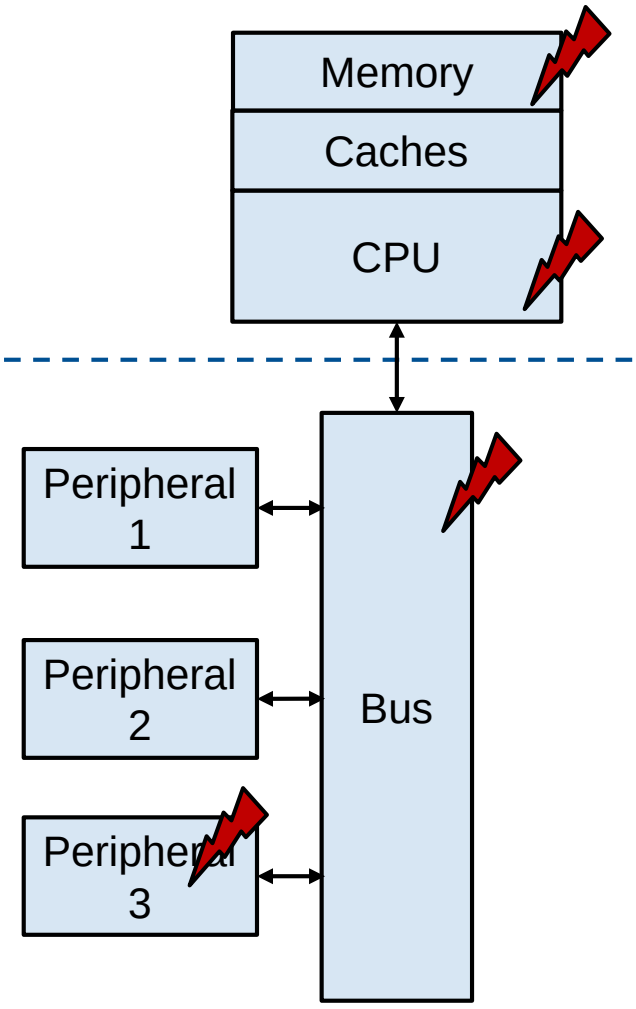
HW-based mechanisms:	SW-based mechanisms inserted by compilers:
• Lockstep • ECC for memory	• Instruction duplication • Variable duplication • Control-flow checking

Mechanisms traditionally added manually by designers:

- Integrity checks
- Checksums
- Readback
- Readtwice

- Error prone
- High design effort

Error Detection and Handling



HW-based mechanisms:	SW-based mechanisms inserted by compilers:
• Lockstep • ECC for memory	• Instruction duplication • Variable duplication • Control-flow checking

Mechanisms traditionally added manually by designers:

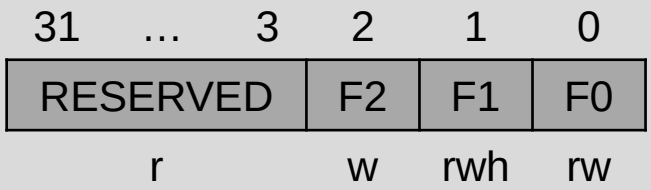
- Integrity checks
- Checksums
- Readback
- Readtwice

- Error prone
- High design effort

SeRoHAL:
Automatically adds error detection mechanisms to hardware abstraction layer (HAL)

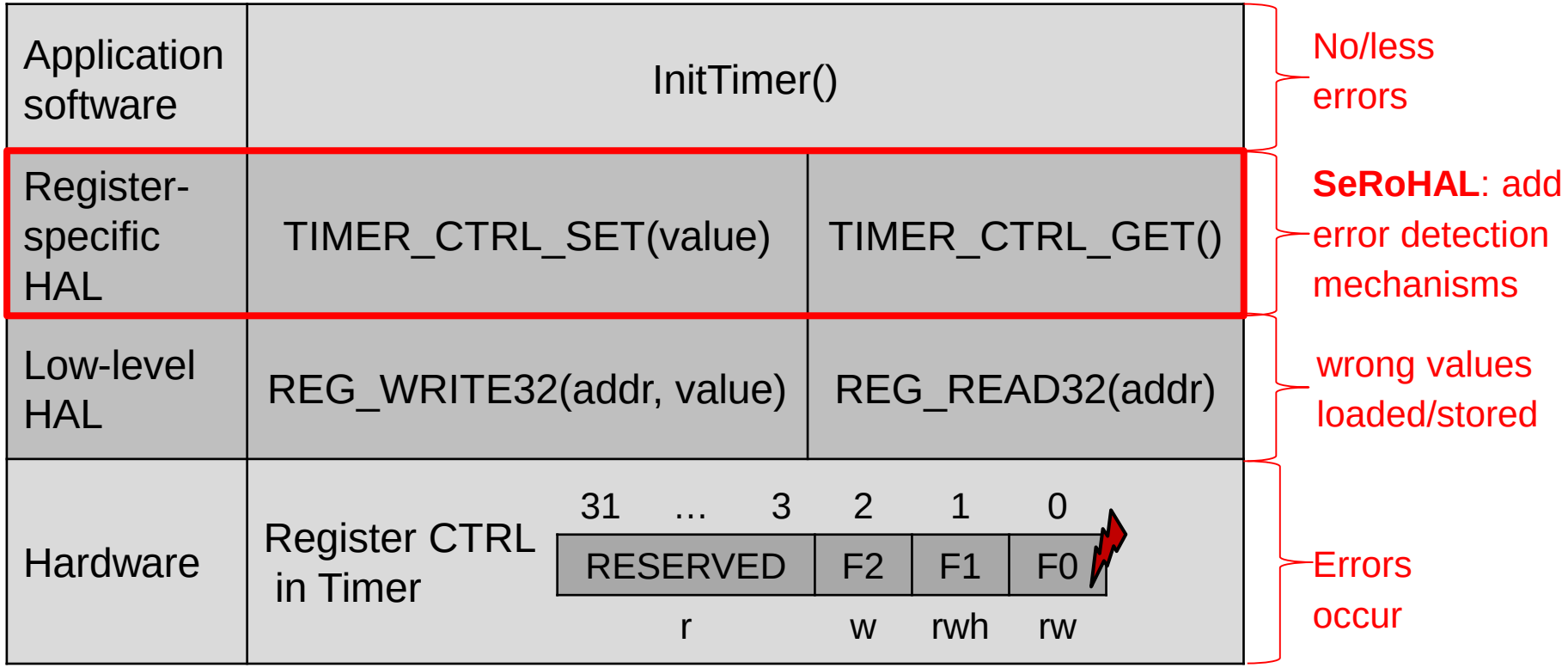
Software Architecture with HAL

HAL: functions that provide read and write access to peripherals registers
+ encapsulate information on hardware architecture, e.g., address of register
+ portability

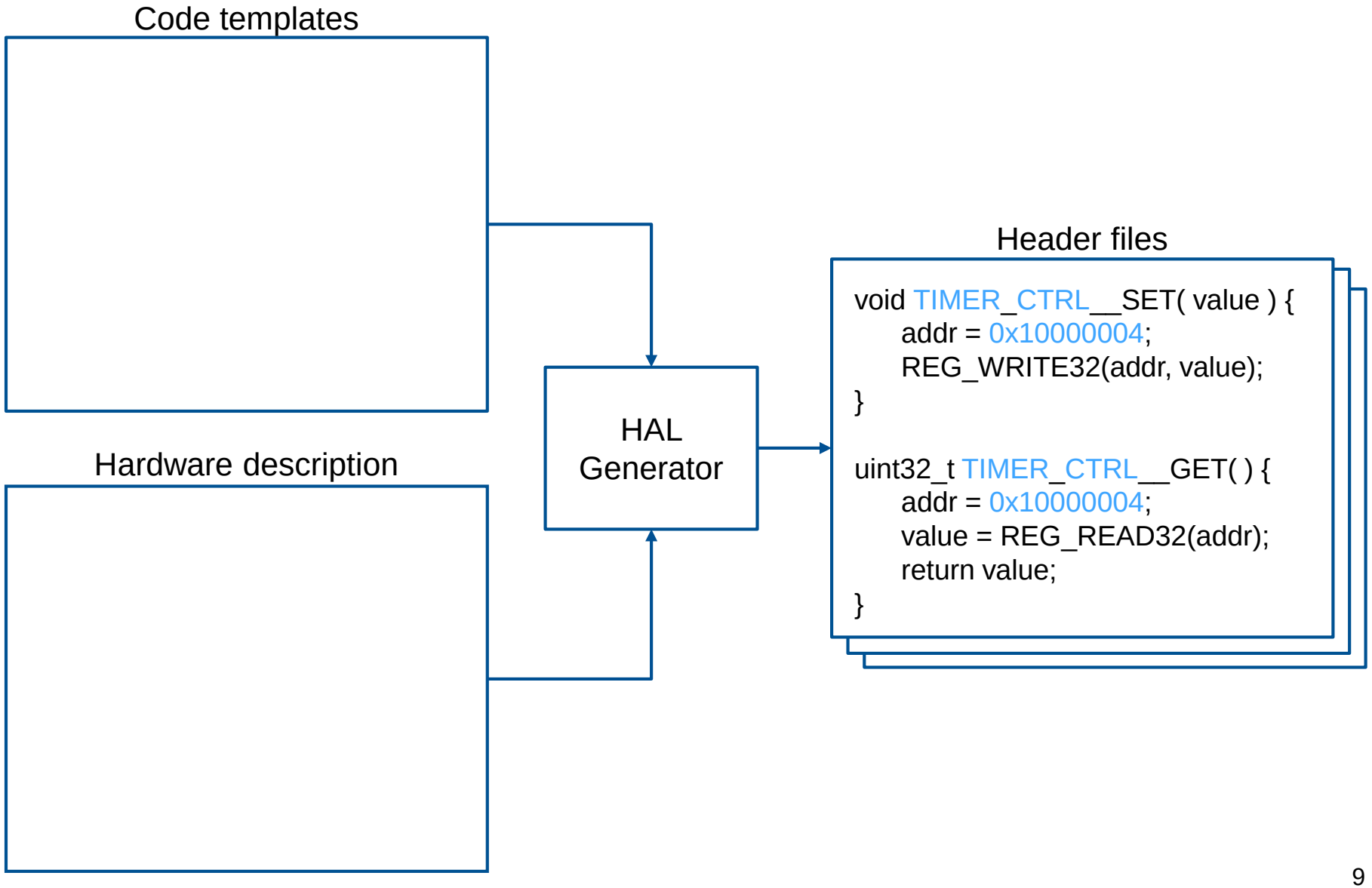
<table>
<tr><td>Application software</td><td colspan="2">InitTimer()</td></tr>
<tr><td>Register-specific HAL</td><td>TIMER_CTRL_SET(value)</td><td>TIMER_CTRL_GET()</td></tr>
<tr><td>Low-level HAL</td><td>REG_WRITE32(addr, value)</td><td>REG_READ32(addr)</td></tr>
<tr><td>Hardware</td><td colspan="2">Register CTRL in Timer

</td></tr>
</table>

Software Architecture with HAL

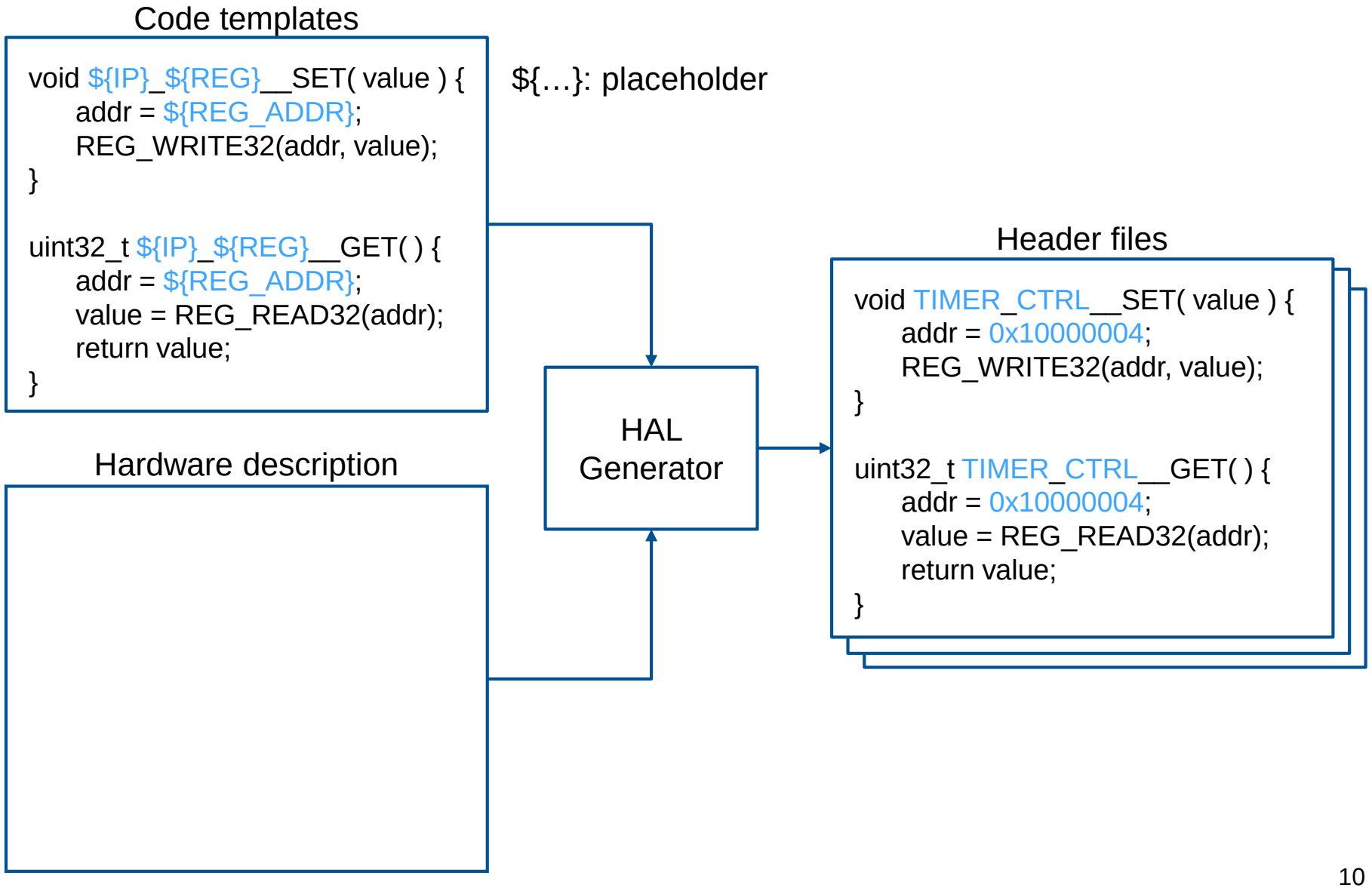
HAL: functions that provide read and write access to peripherals registers
+ encapsulate information on hardware architecture, e.g., address of register
+ portability



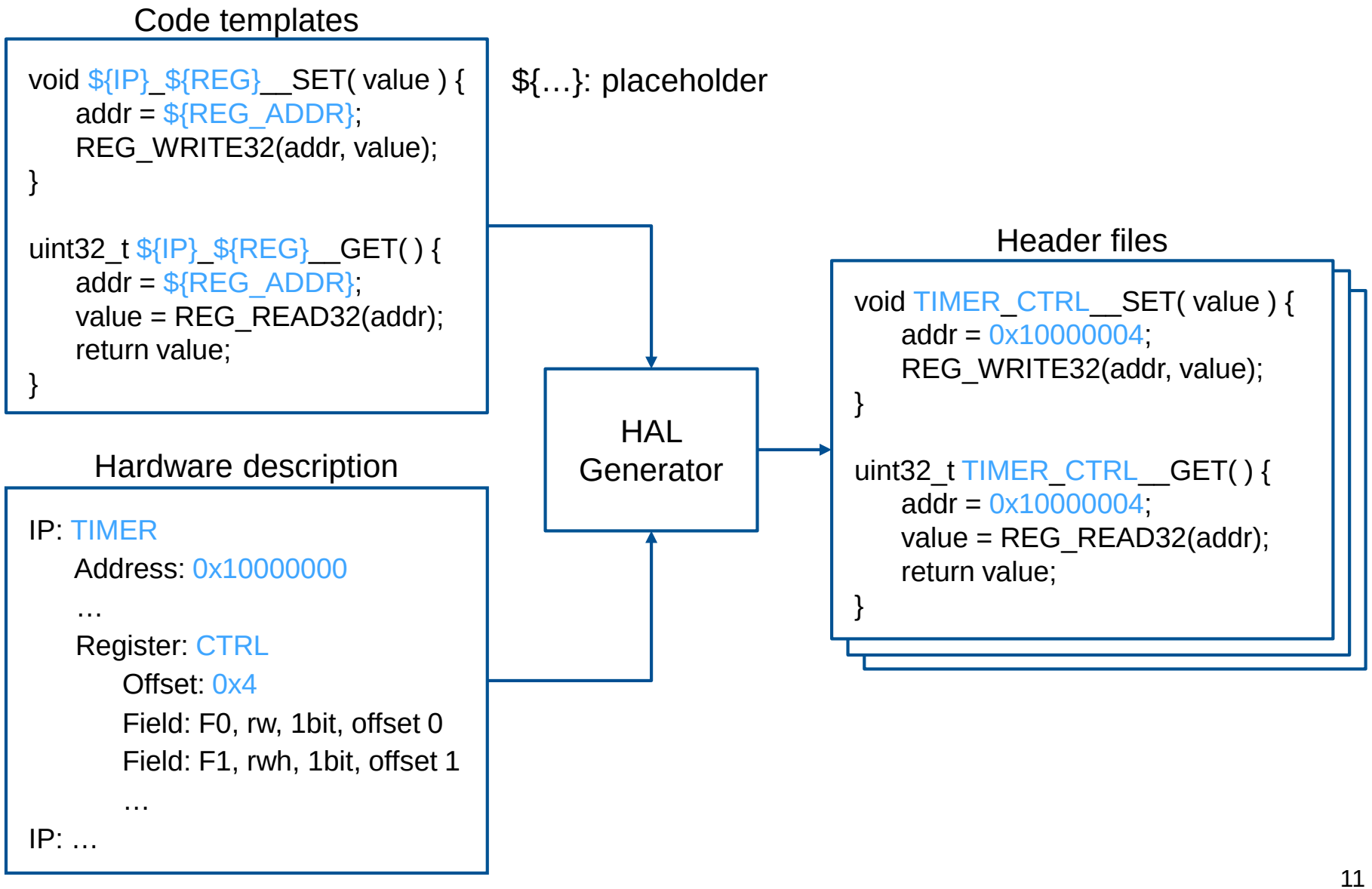
Register-specific HAL Generation



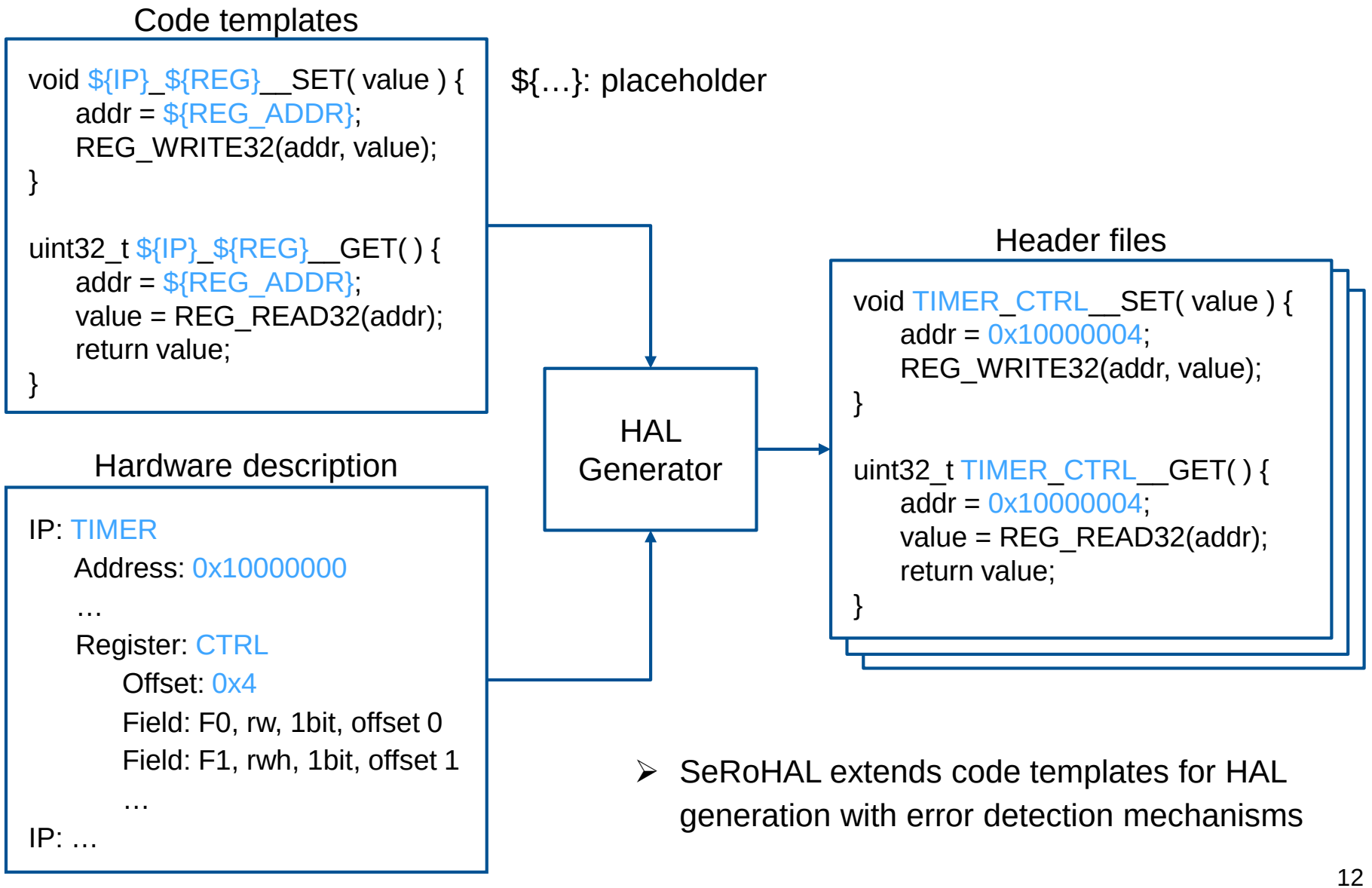
Register-specific HAL Generation



Register-specific HAL Generation



Register-specific HAL Generation



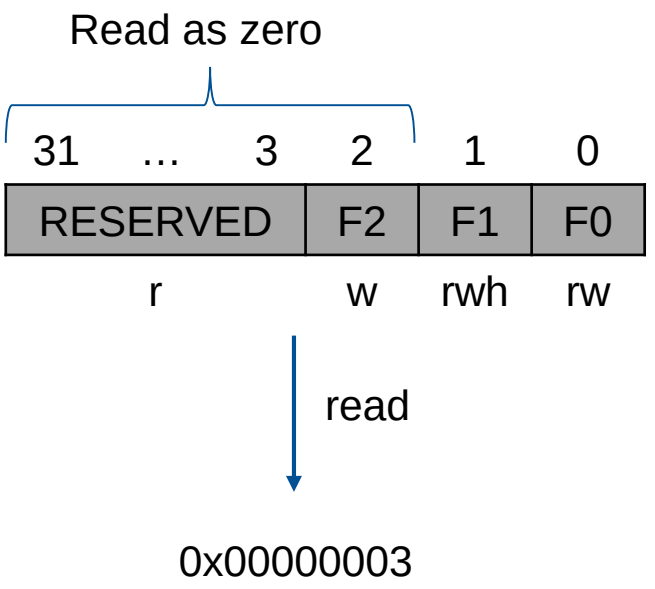
- SeRoHAL extends code templates for HAL generation with error detection mechanisms

Agenda

1. Introduction
2. SeRoHAL
   a) Template-based Generation of Robust Register-specific HAL
      - Masks for irrelevant bits
      - Readtwice
      - Readback
      - Error detection codes (EDC): Word duplication, Checksum, Parity
   b) Selective Protection for Mixed-criticality Systems
3. Results
4. Conclusion

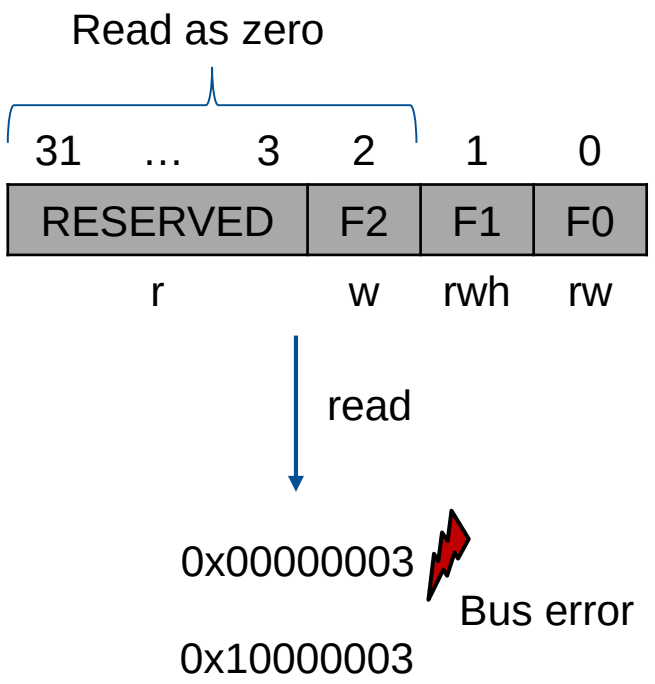
Mask Irrelevant Bits

```
void ${IP}_${REG}__SET( value )
{
    addr = ${REG_ADDR};
    value = value & ${SET_MASK};
    REG_WRITE32(addr, value);



}

uint32_t ${IP}_${REG}__GET( )
{
    addr = ${REG_ADDR};
    value = REG_READ32(addr);
    value = value & ${GET_MASK};



    return value;
}
```



Mask Irrelevant Bits

```
void ${IP}_${REG}__SET( value )
{
    addr = ${REG_ADDR};
    value = value & ${SET_MASK};
    REG_WRITE32(addr, value);



}

uint32_t ${IP}_${REG}__GET( )
{
    addr = ${REG_ADDR};
    value = REG_READ32(addr);
    value = value & ${GET_MASK};



    return value;
}
```

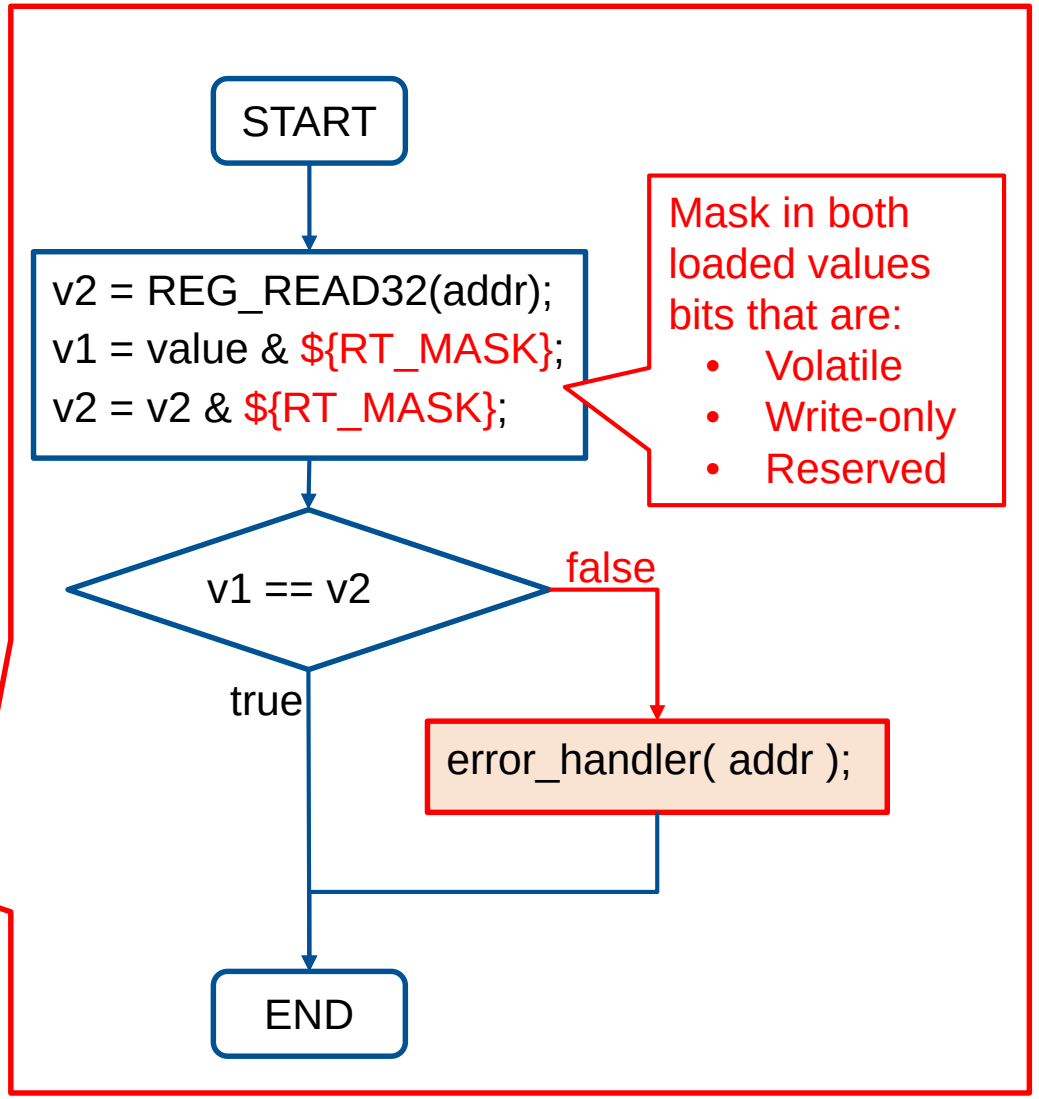


If loaded value is used careless, bus errors in irrelevant bits can cause malfunction

- Mask all reserved bits
- Mask all write-only bits after loading

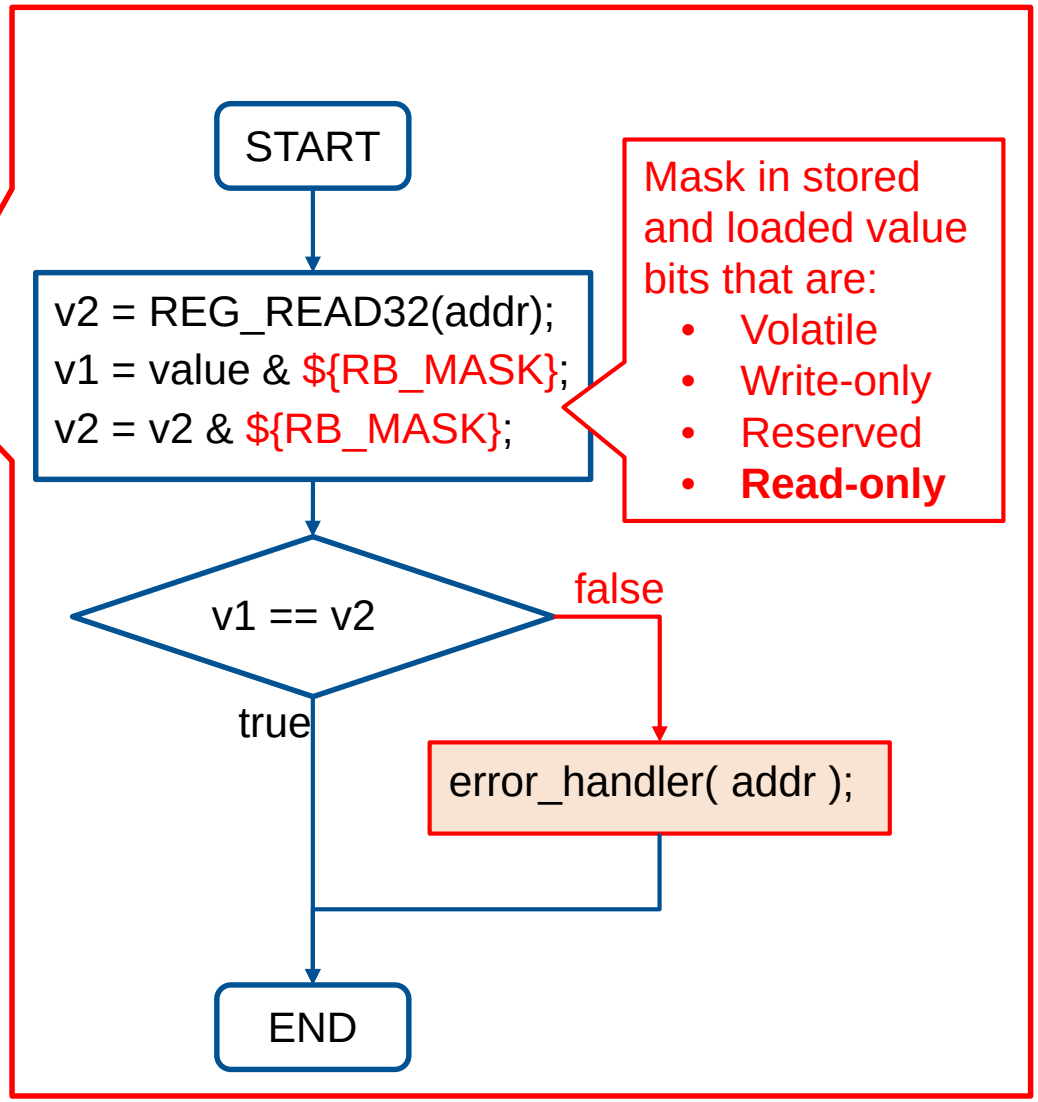
Readtwice

```
void ${IP}_${REG}__SET( value )
{
    addr = ${REG_ADDR};
    value = value & ${SET_MASK};
    REG_WRITE32(addr, value);



}

uint32_t ${IP}_${REG}__GET( )
{
    addr = ${REG_ADDR};
    value = REG_READ32(addr);
    value = value & ${GET_MASK};
    ${READTWICE_PROTECTION}

    return value;
}
```



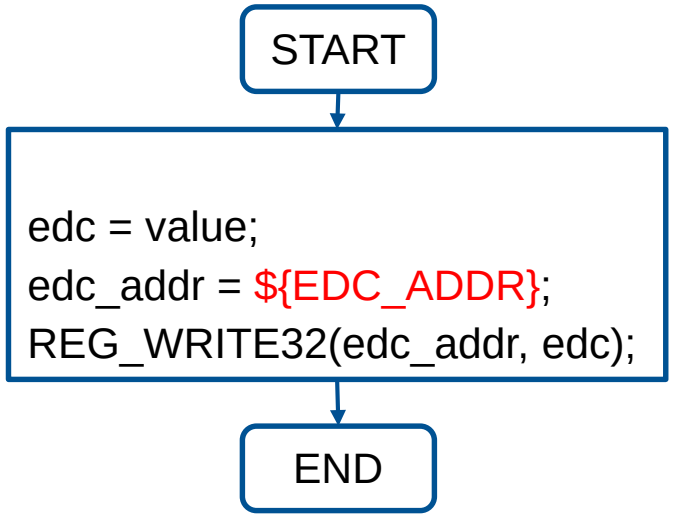
Readback

```
void ${IP}_${REG}__SET( value )
{
    addr = ${REG_ADDR};
    value = value & ${SET_MASK};
    REG_WRITE32(addr, value);
    ${READBACK_PROTECTION}


}

uint32_t ${IP}_${REG}__GET( )
{
    addr = ${REG_ADDR};
    value = REG_READ32(addr);
    value = value & ${GET_MASK};
    ${READTWICE_PROTECTION}

    return value;
}
```



Error Detection Codes (EDCs) – Word Duplication

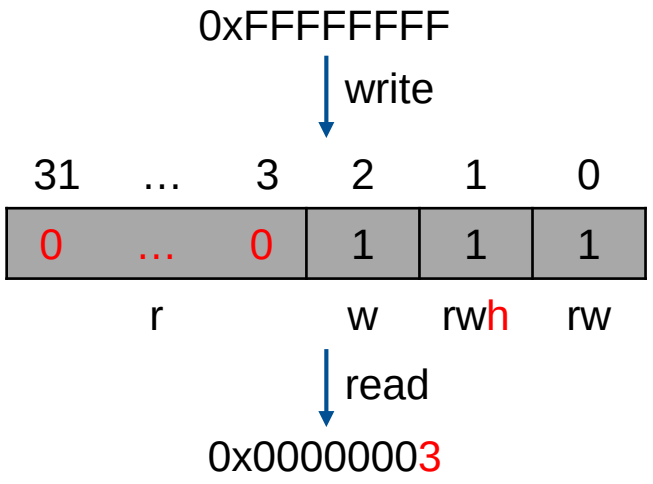
```
void ${IP}_${REG}__SET( value )
{
    addr = ${REG_ADDR};
    value = value & ${SET_MASK};
    REG_WRITE32(addr, value);
    ${READBACK_PROTECTION}
    ${STORE_EDC}
}
```



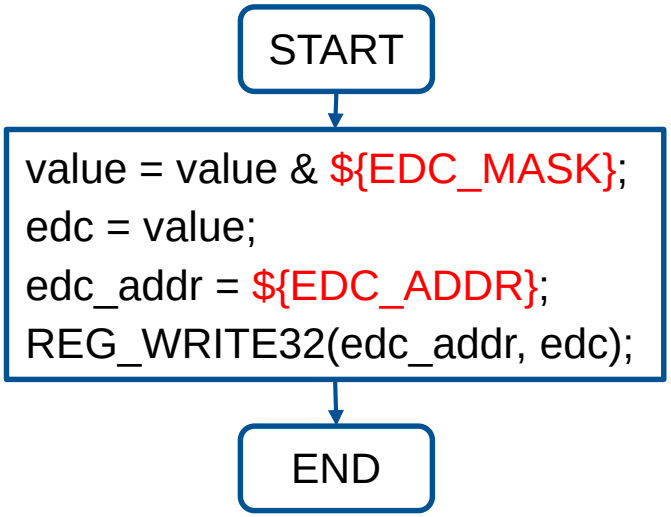
1. Calculate EDC of stored value
2. Store EDC to redundant memory location

Error Detection Codes (EDCs) – Word Duplication

```
void ${IP}_${REG}__SET( value )
{
    addr = ${REG_ADDR};
    value = value & ${SET_MASK};
    REG_WRITE32(addr, value);
    ${READBACK_PROTECTION}
    ${STORE_EDC}
}
```



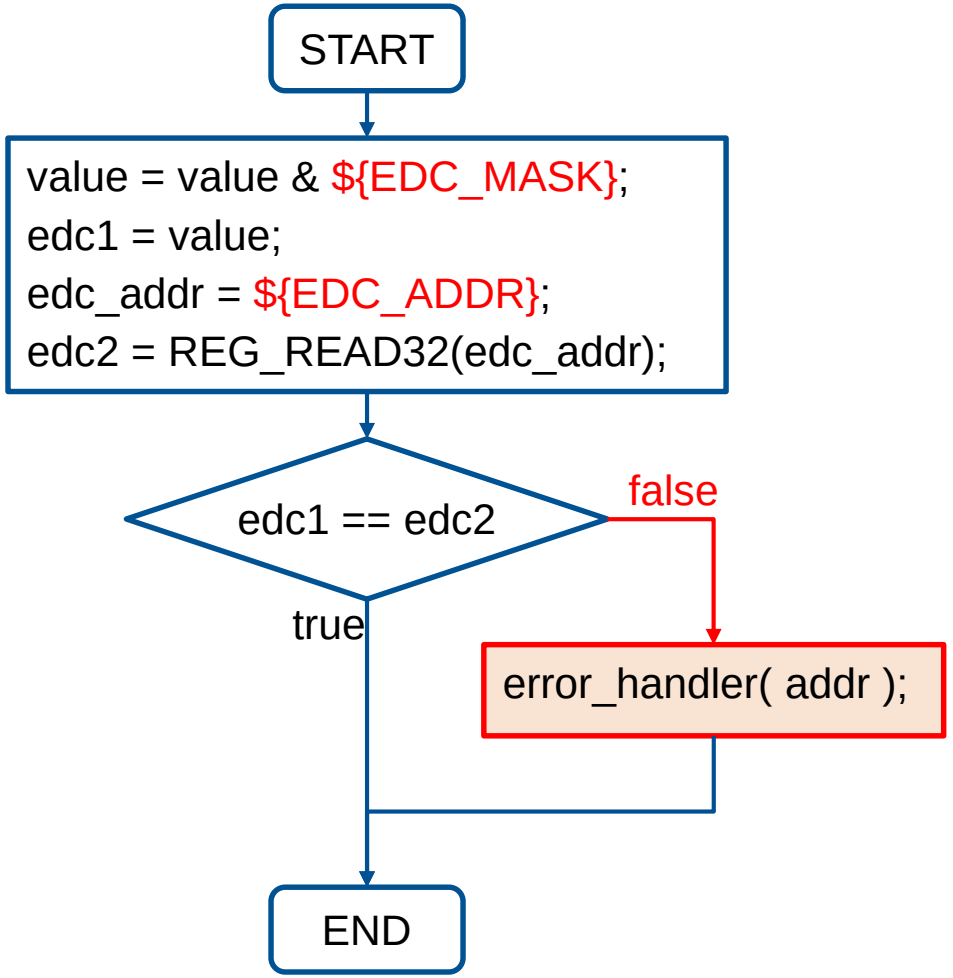
0. Mask not protectable bits in stored value:
    - Read-only bits
    - Reserved bits
    - Write-only bits
    - Volatile bits
1. Calculate EDC of stored value
2. Store EDC to redundant memory location



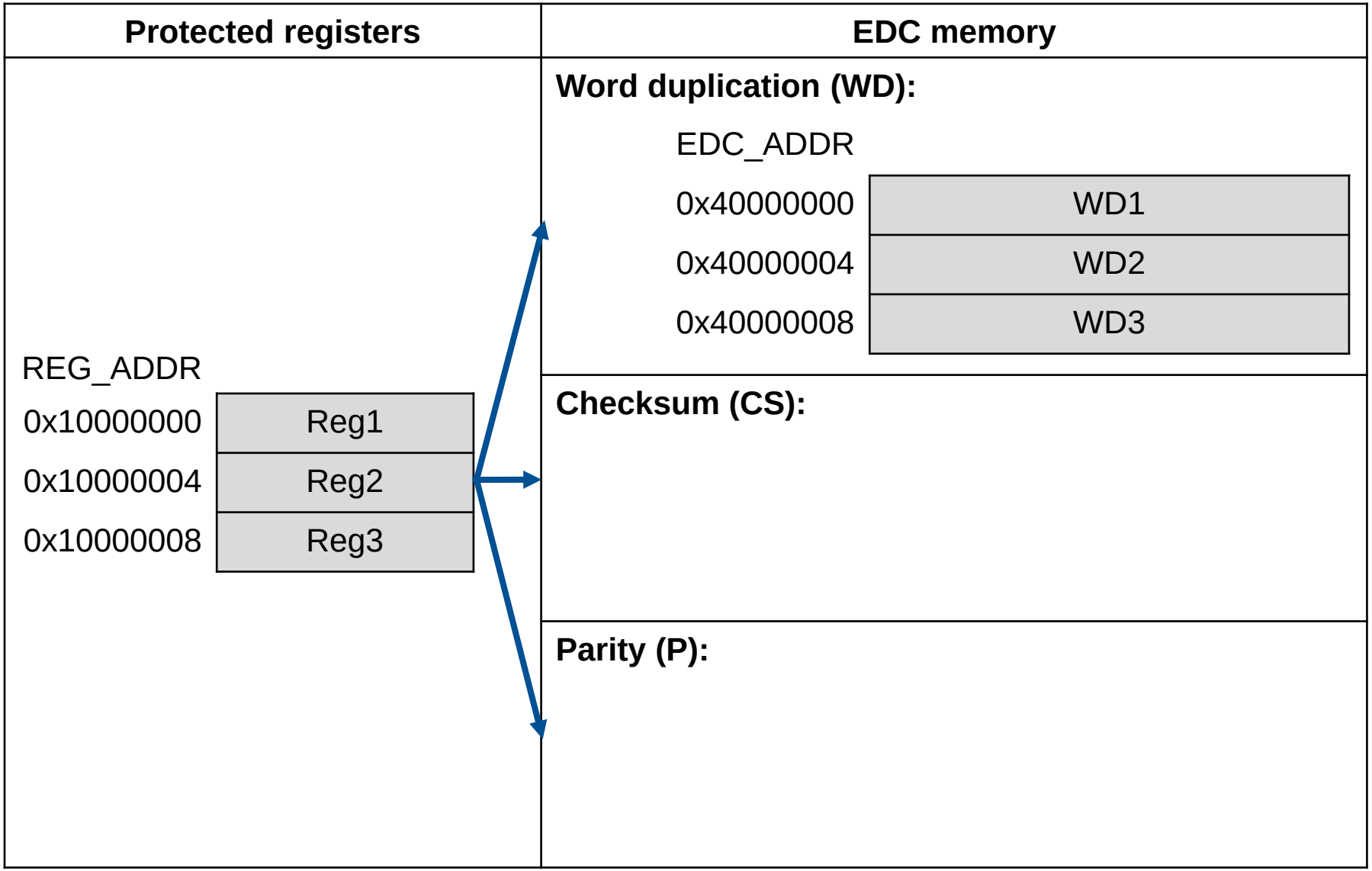
Error Detection Codes (EDCs) – Word Duplication

3. Mask not protectable bits in loaded value:
   - Read-only bits
   - Reserved bits
   - Write-only bits
   - Volatile bits
4. Calculate EDC of value
5. Load EDC from redundant memory location
6. Verify if EDCs are equal

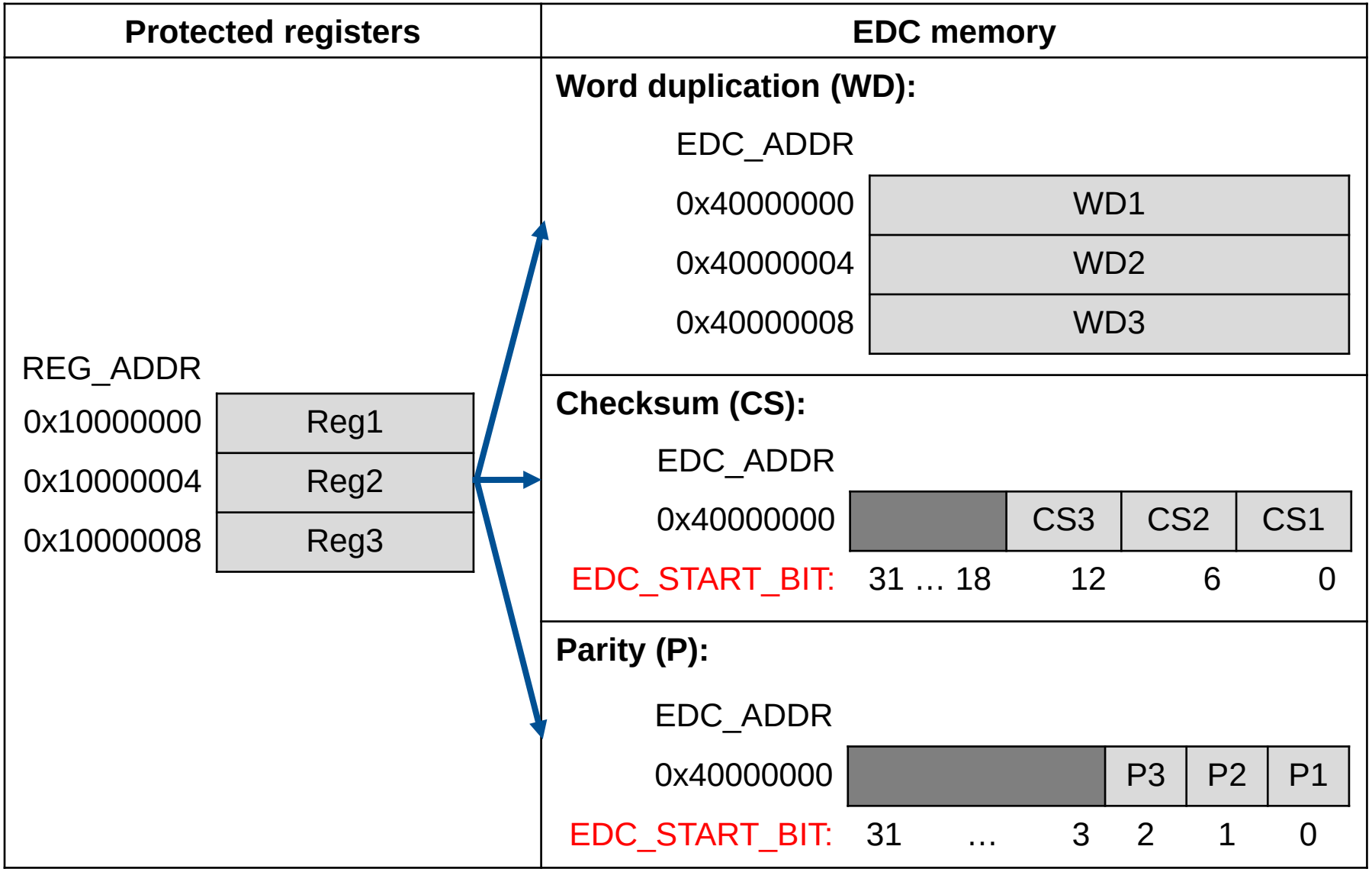
```
uint32_t ${IP}_${REG}__GET( )
{
    addr = ${REG_ADDR};
    value = REG_READ32(addr);
    value = value & ${GET_MASK};
    ${READTWICE_PROTECTION}
    ${LOAD_AND_VERIFY_EDC}
    return value;
}
```



Error Detection Codes – Memory Layout



Error Detection Codes – Memory Layout



Error Detection Codes (EDCs)

	Word duplication	Checksum	Parity
EDC storage overhead	High	Medium	Low
Needs EDC extraction after load ➢ code size overhead	No	Yes	Yes
Needs read-modify-write operation to store EDC ➢ code size and performance overhead	No	Yes	Yes
Error detection capability	High	Medium	Low

Agenda

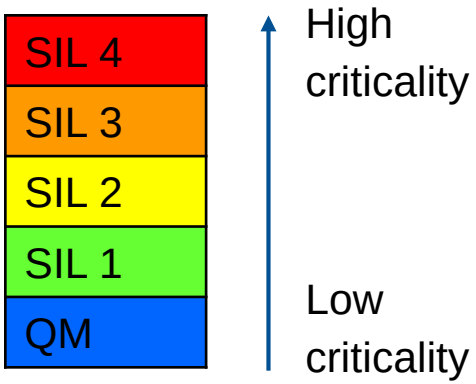
1. Introduction
2. SeRoHAL
   a) Template-based Generation of Robust Register-specific HAL
   b) Selective Protection for Mixed-criticality Systems
3. Results
4. Conclusion

SeRoHAL for Mixed Criticality Systems

Mixed criticality system: executes critical and non-critical tasks

Classification of criticality in international safety standards:

- IEC 61508: safety integrity levels SIL 1-4
- Uncritical: quality management



Protect complete system:

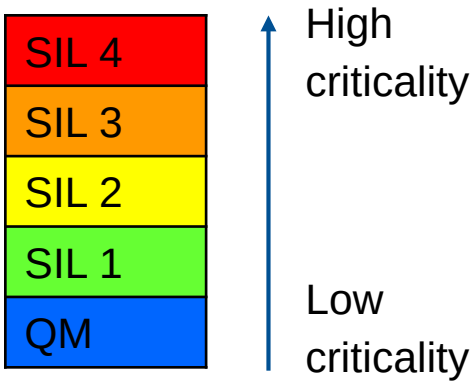
- ➢ High performance and memory overhead

SeRoHAL for Mixed Criticality Systems

Mixed criticality system: executes critical and non-critical tasks

Classification of criticality in international safety standards:

- IEC 61508: safety integrity levels SIL 1-4
- Uncritical: quality management



Protect complete system:

➢ High performance and memory overhead

Select protection according to access criticality:

➢ Avoid unnecessary overhead

Problem:

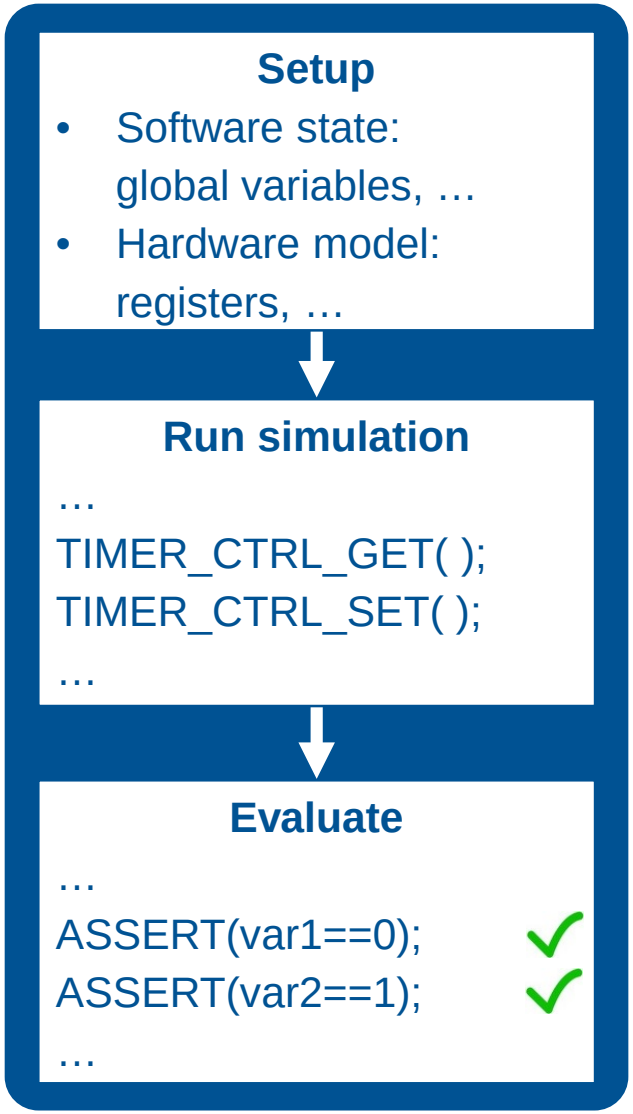
- Criticality levels belong to requirements
- SeRoHAL protects register accesses

➢ Map safety requirement with criticality to hardware accesses

Mapping Criticality to Hardware Accesses

Functional embedded software tests:

- No hardware errors
- Assertions describe requirements



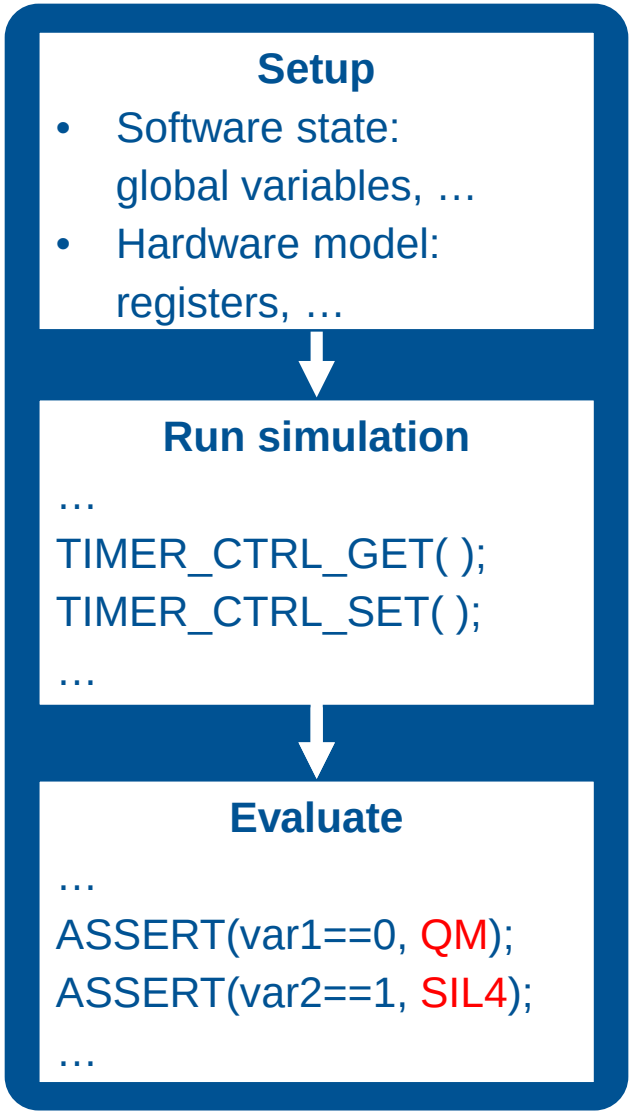
Mapping Criticality to Hardware Accesses

Functional embedded software tests:

- No hardware errors
- Assertions describe requirements

Relate criticality to hardware access:

1. Annotate requirement or criticality to assertions



Mapping Criticality to Hardware Accesses

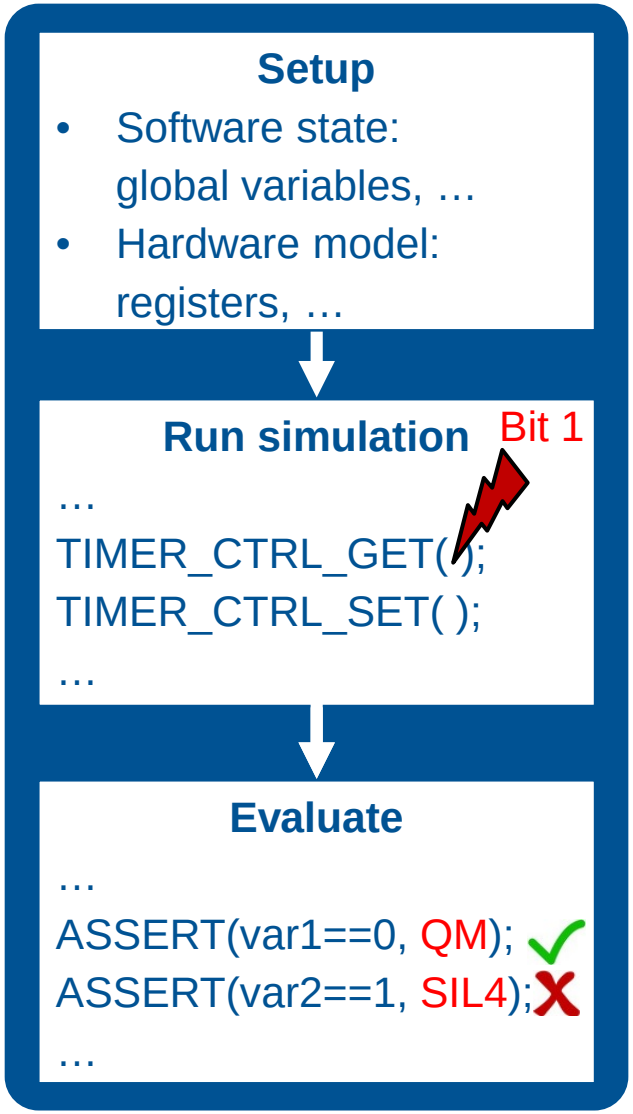
Functional embedded software tests:

- No hardware errors
- Assertions describe requirements

Relate criticality to hardware access:

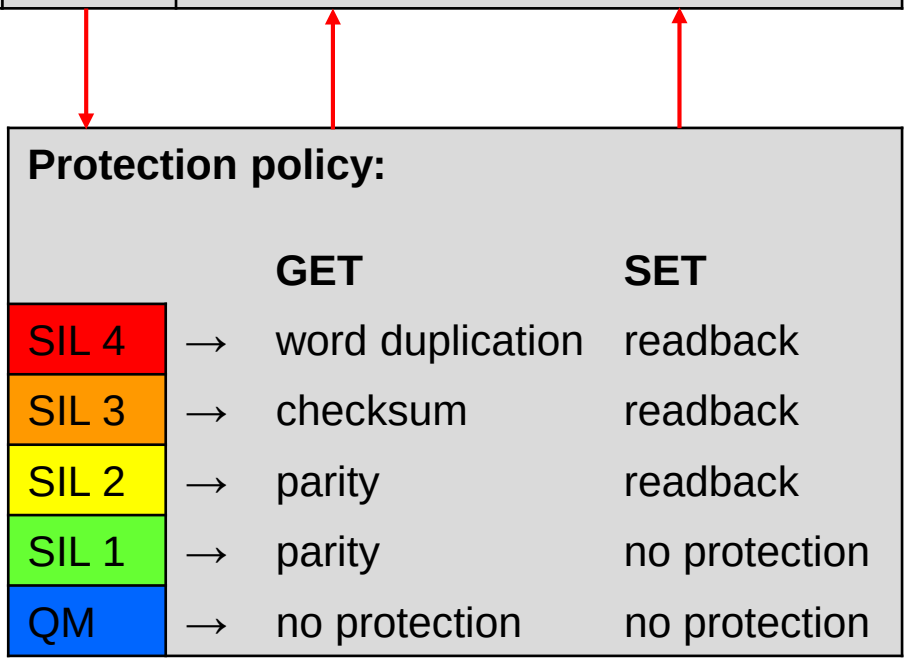
1. Annotate requirement or criticality to assertions
2. Fault injection into register accesses
3. Errors trigger criticality-aware assertions
4. Assign worst-case criticality to hardware accesses:

TIMER_CTRL_GET	SIL4
TIMER_CTRL_SET	QM



Selective Protection

TIMER_CTRL_GET	SIL4	word duplication
TIMER_CTRL_SET	QM	no protection



Agenda

1. Introduction
2. SeRoHAL
   a) Template-based Generation of Robust Register-specific HAL
   b) Selective Protection for Mixed-criticality Systems
3. Results
4. Conclusion

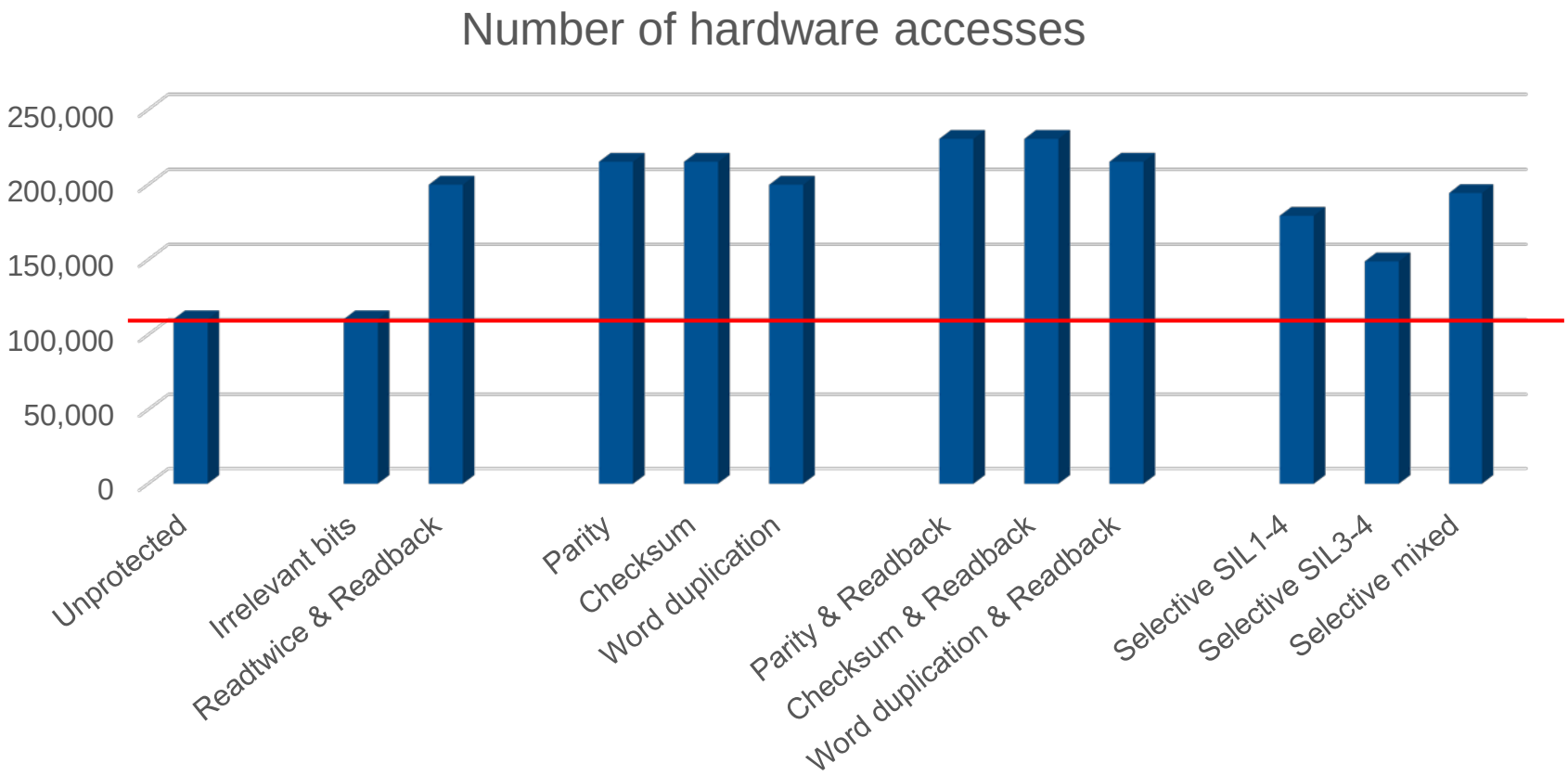
Setup

- Generated 12 HALs for example software:
  - Robot arm control
  - Implemented in C
  - Executes on XMC4500

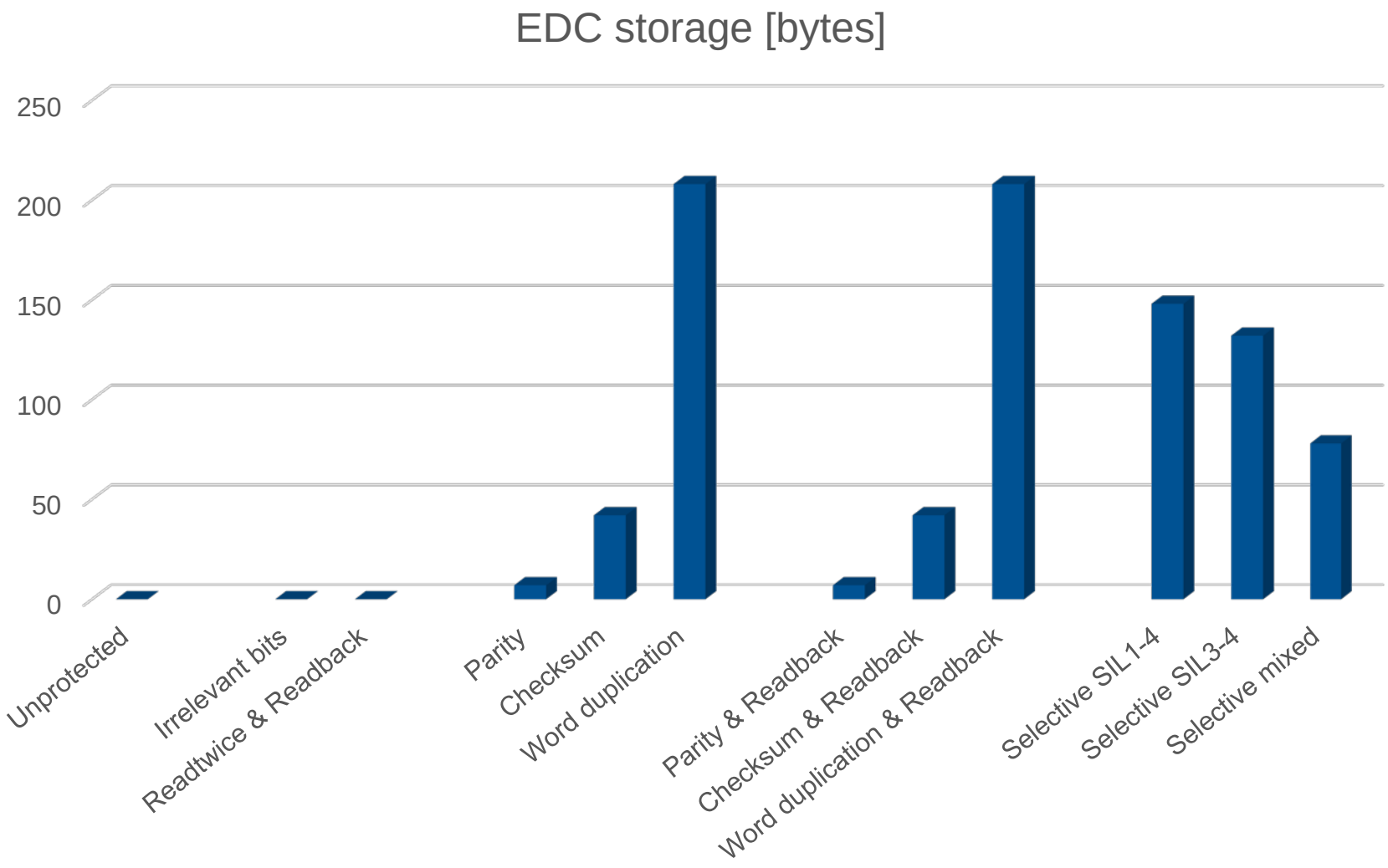
- Implemented 312 functional software tests
  - Access 52 registers
  - 1,084 criticality-aware assertions
  - Random assignment of criticality levels QM and SILs 1-4 to assertions
  - Inject all possible 691,680 single-bit and double-bit bus errors

Overhead – Hardware accesses

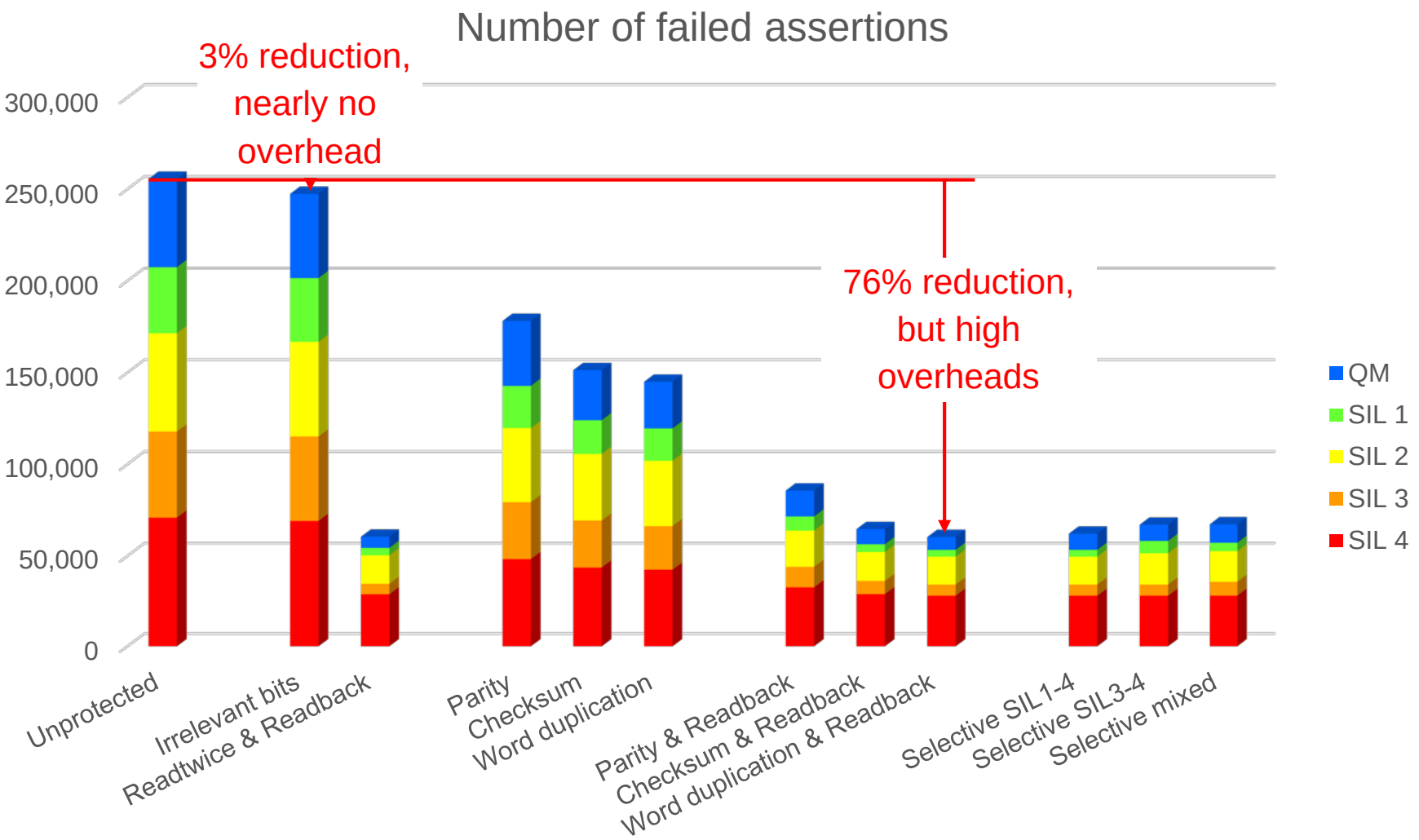
- Number of accesses performed during fault free full system simulation
- Indicator for performance overhead



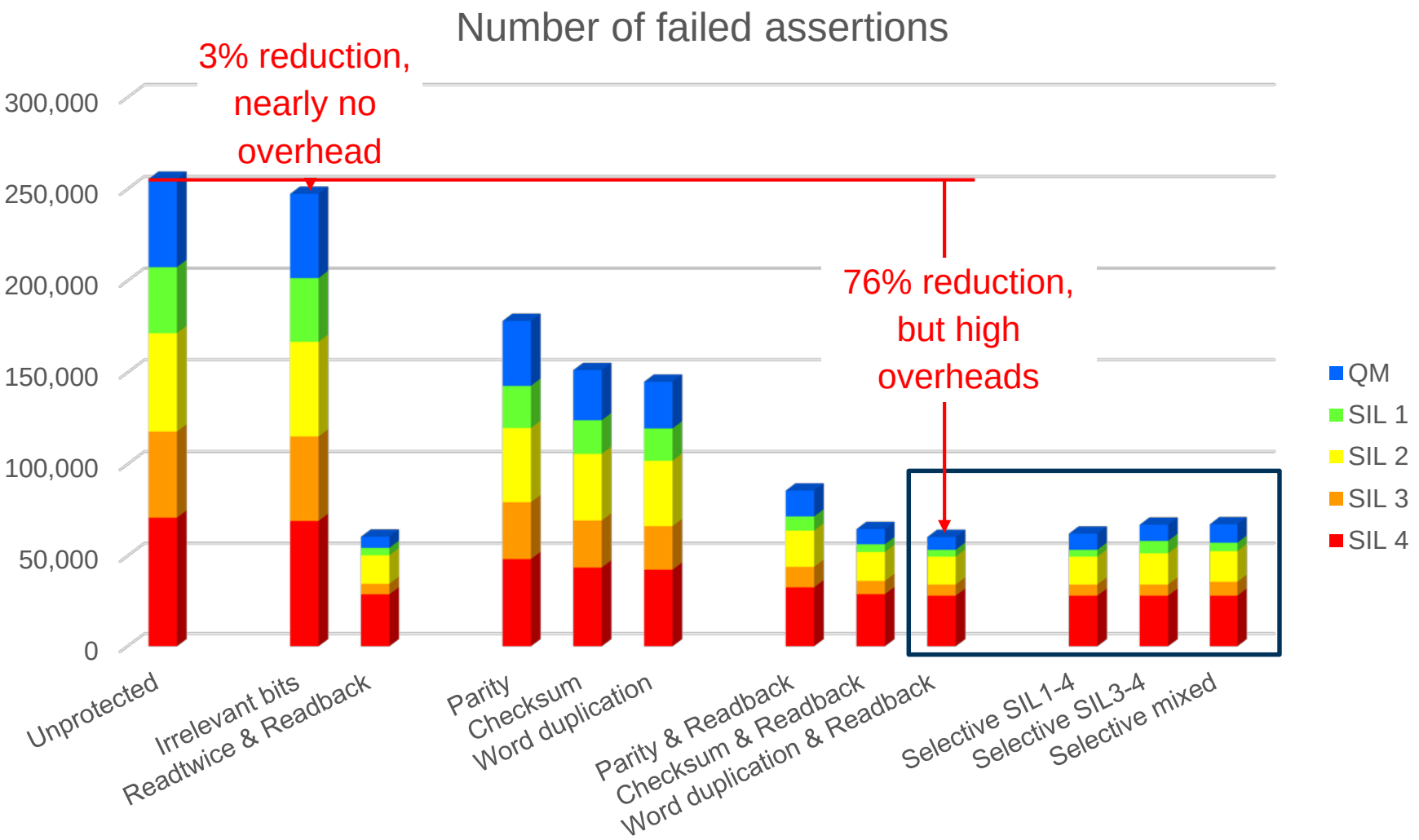
Overhead – RAM



Robustness


Number of failed assertions
3% reduction, nearly no overhead
76% reduction, but high overheads
300,000
250,000
200,000
150,000
100,000
50,000
0
Unprotected
Irrelevant bits Readtwice & Readback
Parity
Checksum
Word duplication
Parity & Readback
Checksum & Readback
Word duplication & Readback
Selective SIL1-4
Selective SIL3-4
Selective mixed
QM
SIL 1
SIL 2
SIL 3
SIL 4

Robustness


Number of failed assertions
3% reduction, nearly no overhead
76% reduction, but high overheads
300,000
250,000
200,000
150,000
100,000
50,000
0
Unprotected
Irrelevant bits Readtwice & Readback
Parity
Checksum
Word duplication
Parity & Readback
Checksum & Readback
Word duplication & Readback
Selective SIL1-4
Selective SIL3-4
Selective mixed
QM
SIL 1
SIL 2
SIL 3
SIL 4

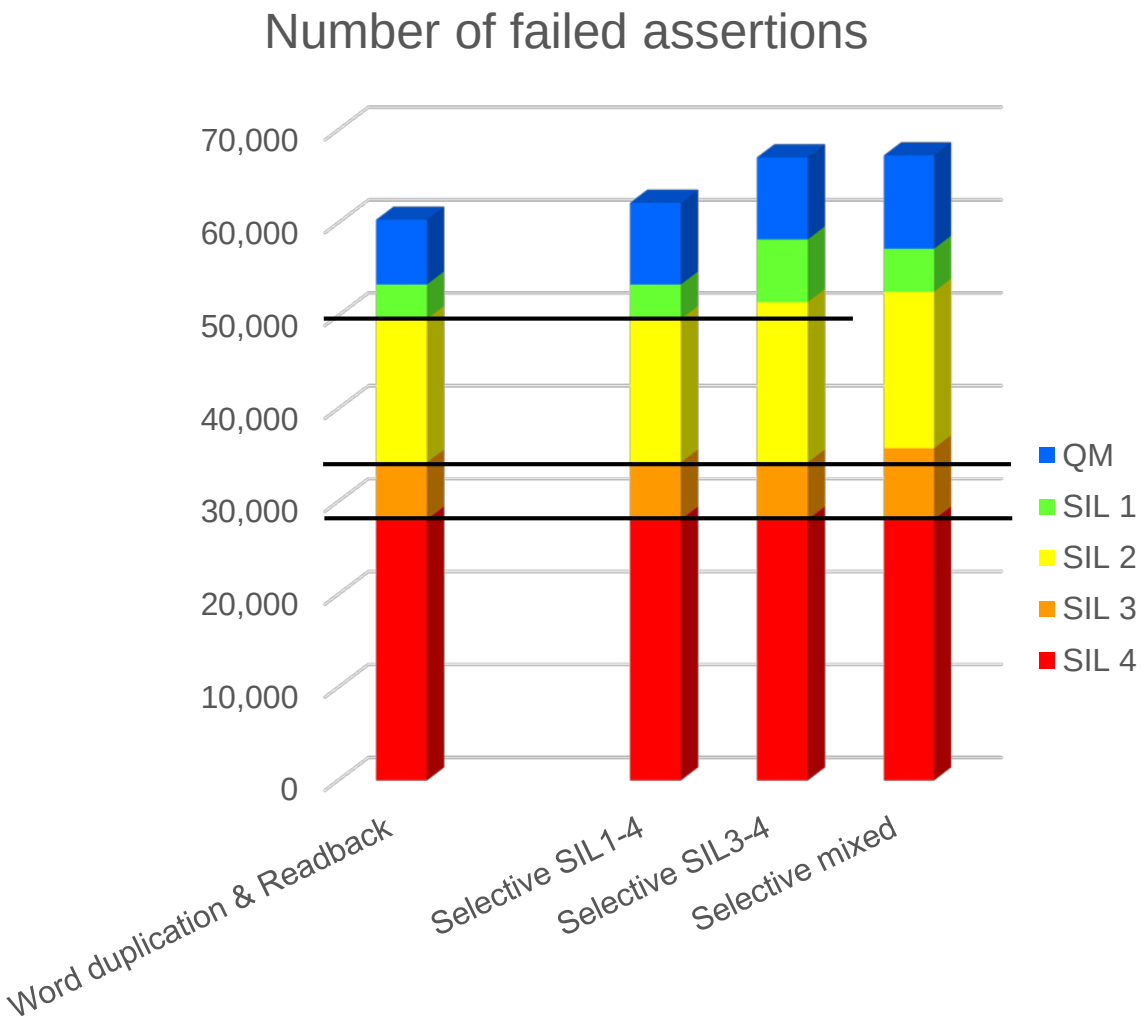
Robustness – Selective Protection

Low criticality

- Less or no protection
- More failures

High criticality

- Keep strong protection
- No additional failures



Agenda

1. Introduction
2. SeRoHAL
   a) Template-based Generation of Robust Register-specific HAL
   b) Selective Protection for Mixed-criticality Systems
3. Results
4. Conclusion

Conclusion

Robust HAL:

- Automatic generation from code templates
- 6 safety mechanisms have been enhanced for protecting peripheral registers and have been implemented
- Avoids up to 76% of all failures
- Induces high overheads

Conclusion

Robust HAL:

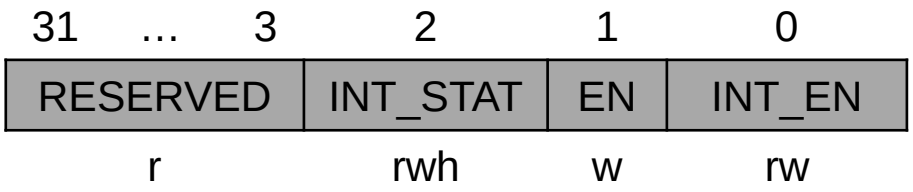
- Automatic generation from code templates
- 6 safety mechanisms have been enhanced for protecting peripheral registers and have been implemented
- Avoids up to 76% of all failures
- Induces high overheads

Selective protection:

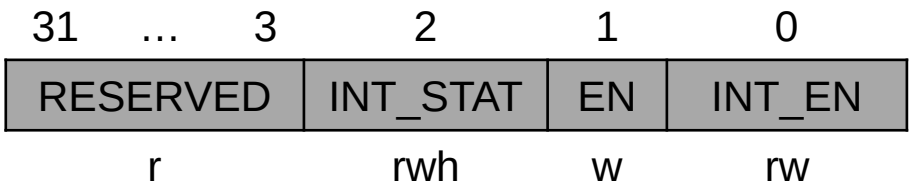
- Selects weaker or no error detection mechanisms for less critical accesses
- Reduces overhead
- Optimal protection policy must be chosen carefully depending on:
  - Performance constraints
  - RAM size constraints
  - ROM size constraints
  - Application properties
  - Fault tolerance requirements

Backup

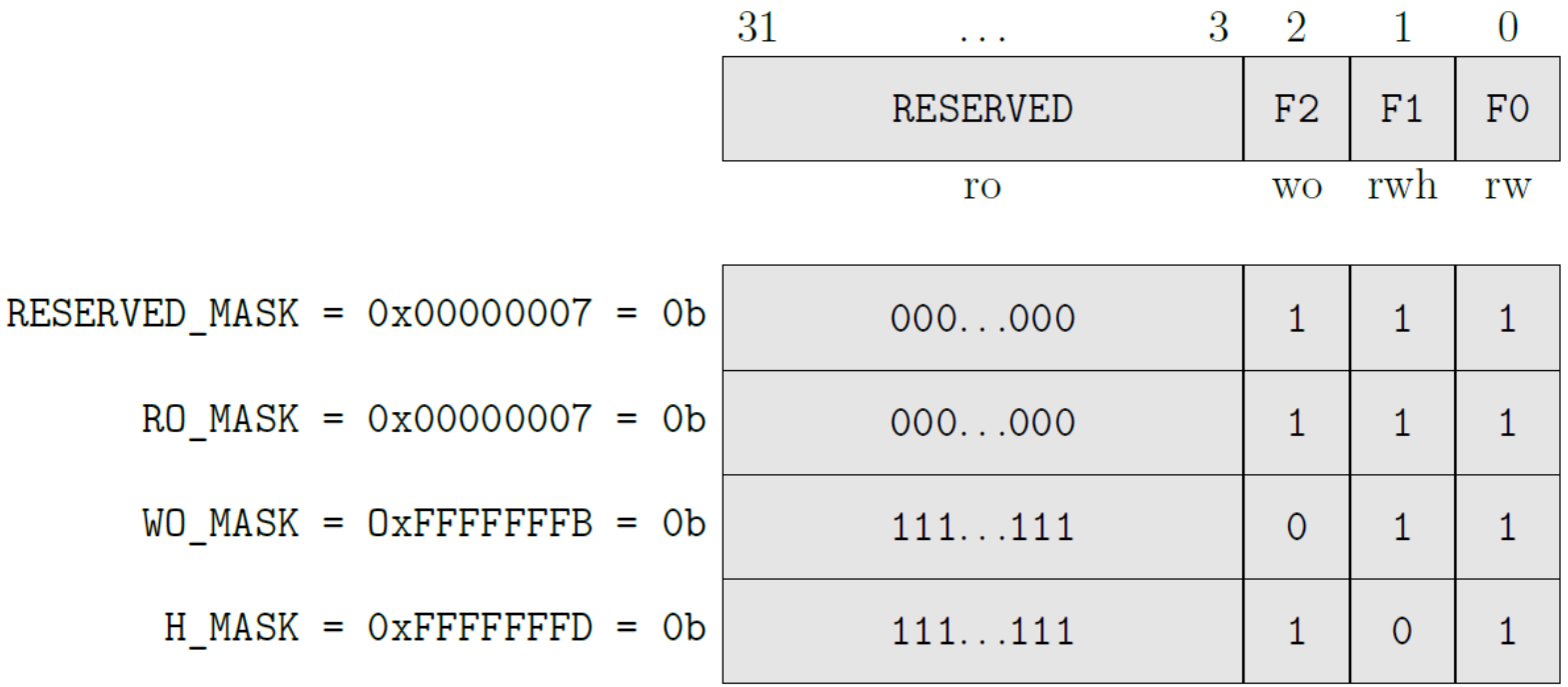
```
...
<peripheral>
   <name> TIMER </name>
   <baseAddress> 0x10000000 </baseAddress>
   ...
   <registers>
      <register>
         <name> CTRL </name>
         <description> Timer control register. </description>
         <addressOffset> 0x000 </addressOffset>
         <size> 32 </size>
         <resetValue> 0x00000000 </resetValue>
         ...
         <fields>
            <field>
               <name> INT_EN </name>
               <description> Interrupt enable flag </description>
               <bitOffset> 0 </bitOffset>
               <bitWidth> 1 </bitWidth>
               <access> read/write </access>
            </field>
```



```
            <field>
                <name> EN </name>
                <description> Timer enable flag </description>
                <bitOffset> 1 </bitOffset>
                <bitWidth> 1 </bitWidth>
                <access> write-only </access>
            </field>
            <field>
                <name> INT_STAT </name>
                <description> Interrupt status flag </description>
                <bitOffset> 2 </bitOffset>
                <bitWidth> 1 </bitWidth>
                <access> read/write </access>
                <volatile> true </volatile>
            </field>
        </fields>
    </register>
  ...
  </registers>
</peripheral>
...
```



Masks


31 . . . 3 2 1 0
RESERVED F2 F1 F0
ro wo rwh rw
RESERVED_MASK = 0x00000007 = 0b 000...000 1 1 1
RO_MASK = 0x00000007 = 0b 000...000 1 1 1
WO_MASK = 0xFFFFFFFB = 0b 111...111 0 1 1
H_MASK = 0xFFFFFFFD = 0b 111...111 1 0 1

Error Detection Codes (EDCs)

```
void ${IP}_${REG}__SET( value )
{
    addr = ${REG_ADDR};
    value = value & ${SET_MASK};
    REG_WRITE32(addr, value);
    ${READBACK_PROTECTION}
    ${STORE_EDC}
}

uint32_t ${IP}_${REG}__GET( )
{
    addr = ${REG_ADDR};
    value = REG_READ32(addr);
    value = value & ${GET_MASK};
    ${READTWICE_PROTECTION}
    ${LOAD_AND_VERIFY_EDC}
    return value;
}
```

Basic functionality:

When storing a value:

1) Calculate EDC of value to be stored
2) Store EDC to redundant memory location

When loading a value:

3) Calculate EDC of loaded value
4) Load EDC from redundant memory location
5) Verify if EDCs are equal

Error Detection Codes (EDCs) – Parity

Load access:

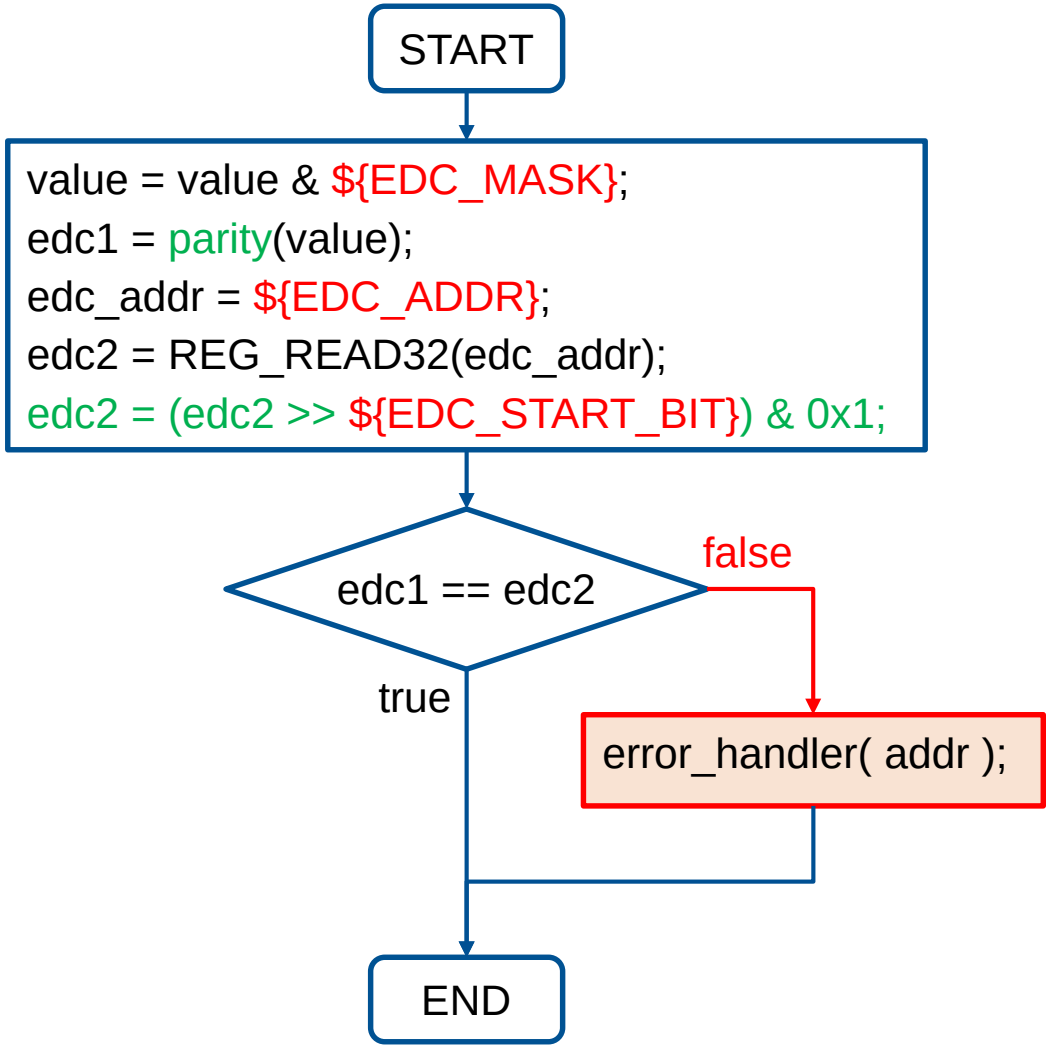
3a) Mask not protectable bits in value:
- Read-only bits
- Reserved bits
- Write-only bits
- Volatile bits

3b) Calculate EDC of value

4) Load EDC word from redundant memory location

4b) Extract parity bit from loaded EDC word

3) Verify if EDCs are equal



Error Detection Codes (EDCs) – Parity

Store access:

1a) Mask not protectable bits in value:
- Read-only bits
- Reserved bits
- Write-only bits
- Volatile bits

1b) Calculate EDC of value to be stored

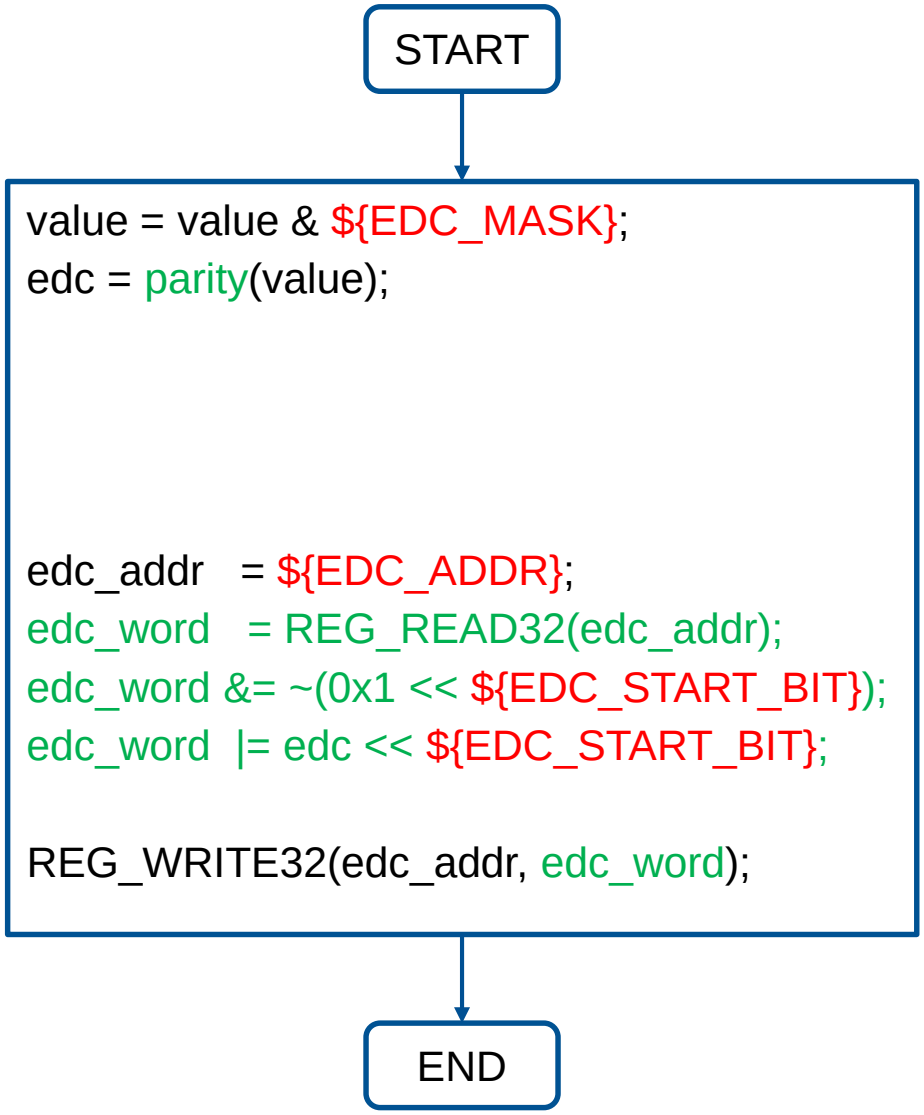
2a) Load EDC word from memory location

2b) Mask old EDC within EDC word

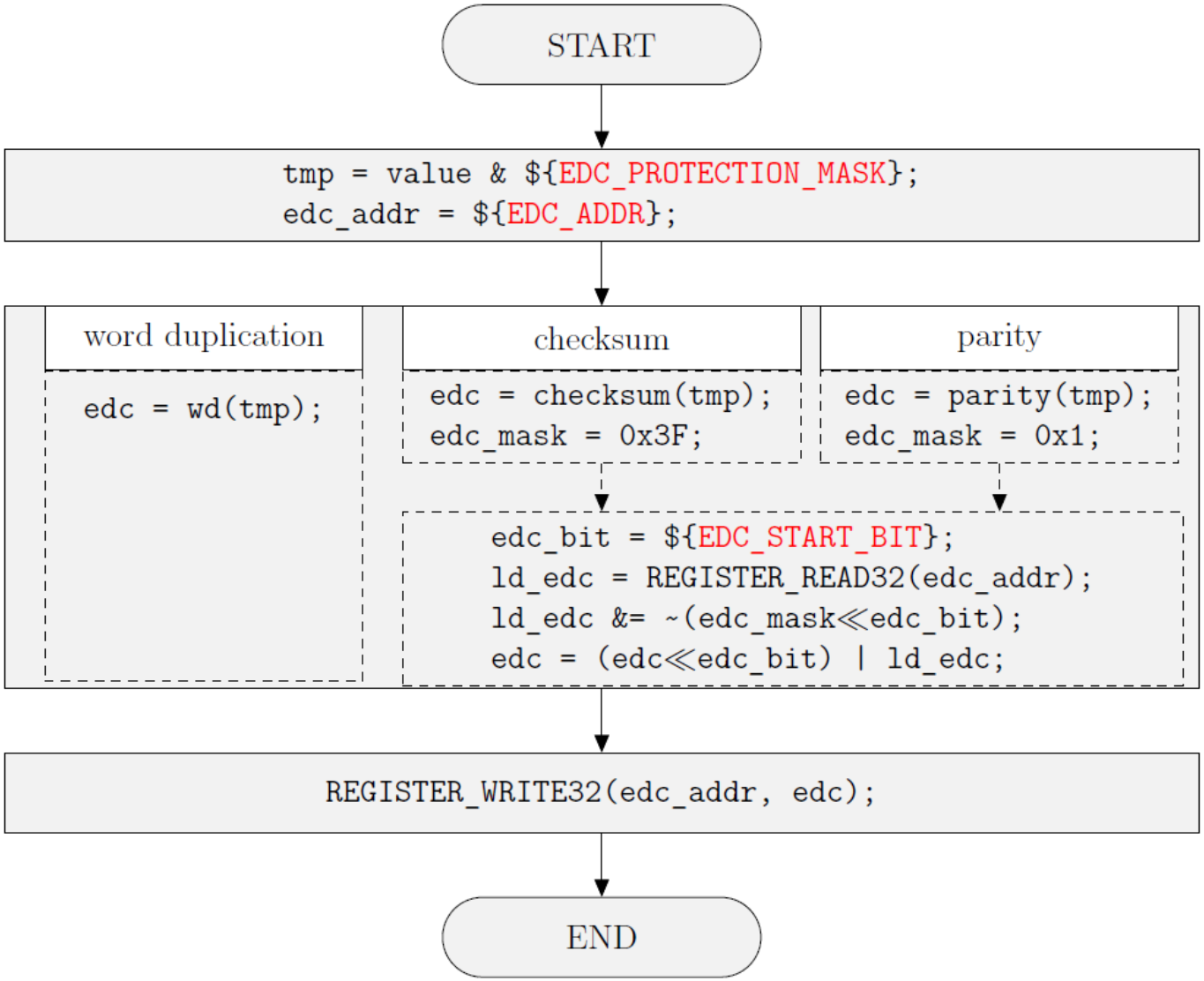
2b) Set new EDC within EDC word

2c) Store EDC to redundant memory location

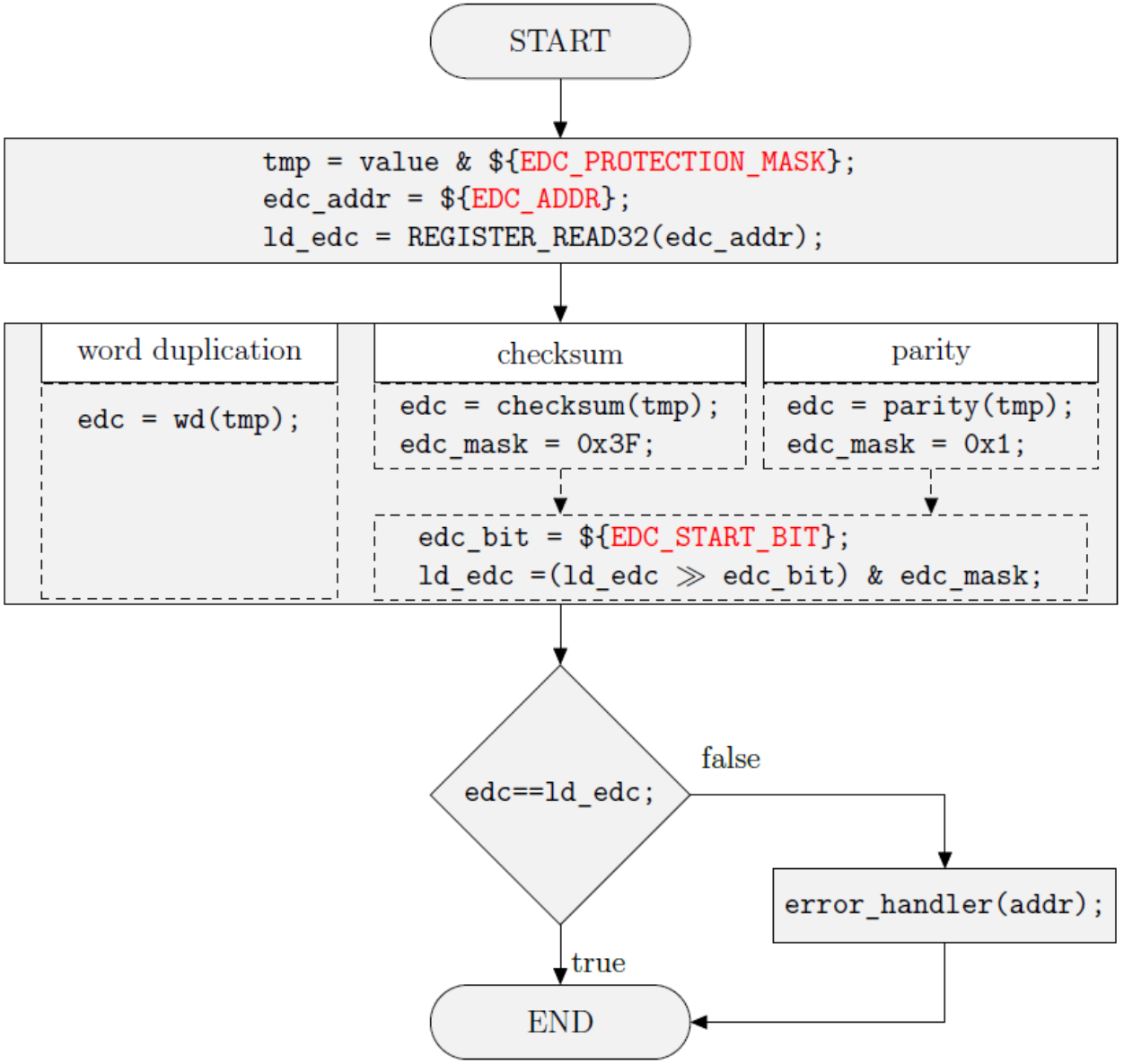
Checksum handled similar to parity



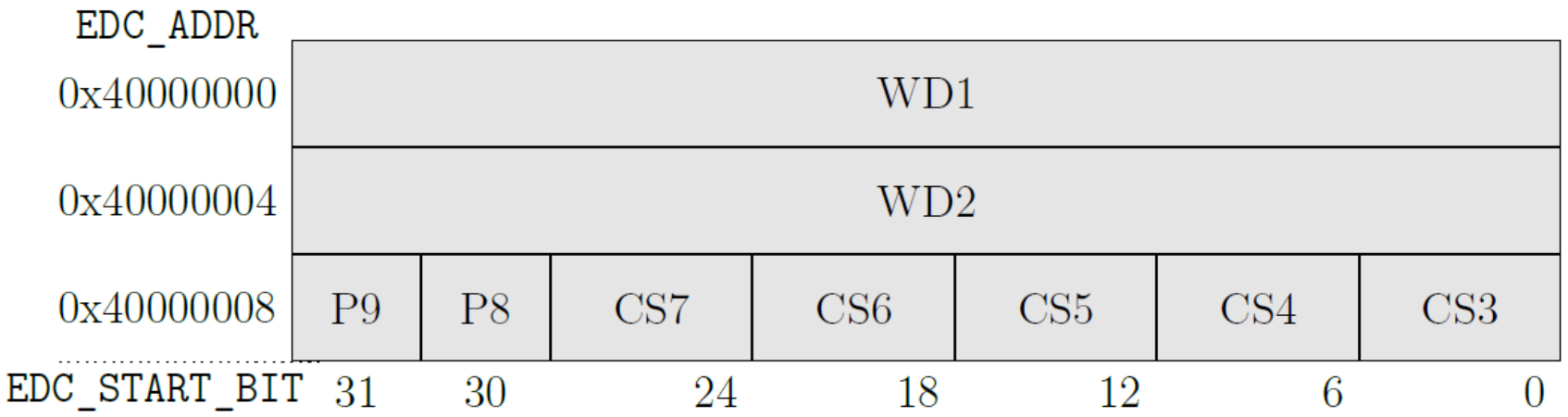
Store EDC


START
tmp = value & ${EDC_PROTECTION_MASK};
edc_addr = ${EDC_ADDR};
word duplication
edc = wd(tmp);
checksum
edc = checksum(tmp);
edc_mask = 0x3F;
parity
edc = parity(tmp);
edc_mask = 0x1;
edc_bit = ${EDC_START_BIT};
ld_edc = REGISTER_READ32(edc_addr);
ld_edc &= ~(edc_mask≪edc_bit);
edc = (edc≪edc_bit) | ld_edc;
REGISTER_WRITE32(edc_addr, edc);
END

Load EDC


START
tmp = value & ${EDC_PROTECTION_MASK};
edc_addr = ${EDC_ADDR};
ld_edc = REGISTER_READ32(edc_addr);
word duplication
edc = wd(tmp);
checksum
edc = checksum(tmp);
edc_mask = 0x3F;
parity
edc = parity(tmp);
edc_mask = 0x1;
edc_bit = ${EDC_START_BIT};
ld_edc =(ld_edc ≫ edc_bit) & edc_mask;
edc==ld_edc;
false
error_handler(addr);
true
END

EDC memory layout



Overheads induced by protection mechanisms

Protection mechanism	RAM overhead per l-bit register [bits]	Additional HW accesses per load access	Additional HW accesses per store access
Irrelevant bit mask	0	0	0
Readback	0	0	1
Readtwice	0	1	0
Parity	1	1	2
Checksum	$\lceil log_2(l+1) \rceil$	1	2
Word duplication	l	1	1

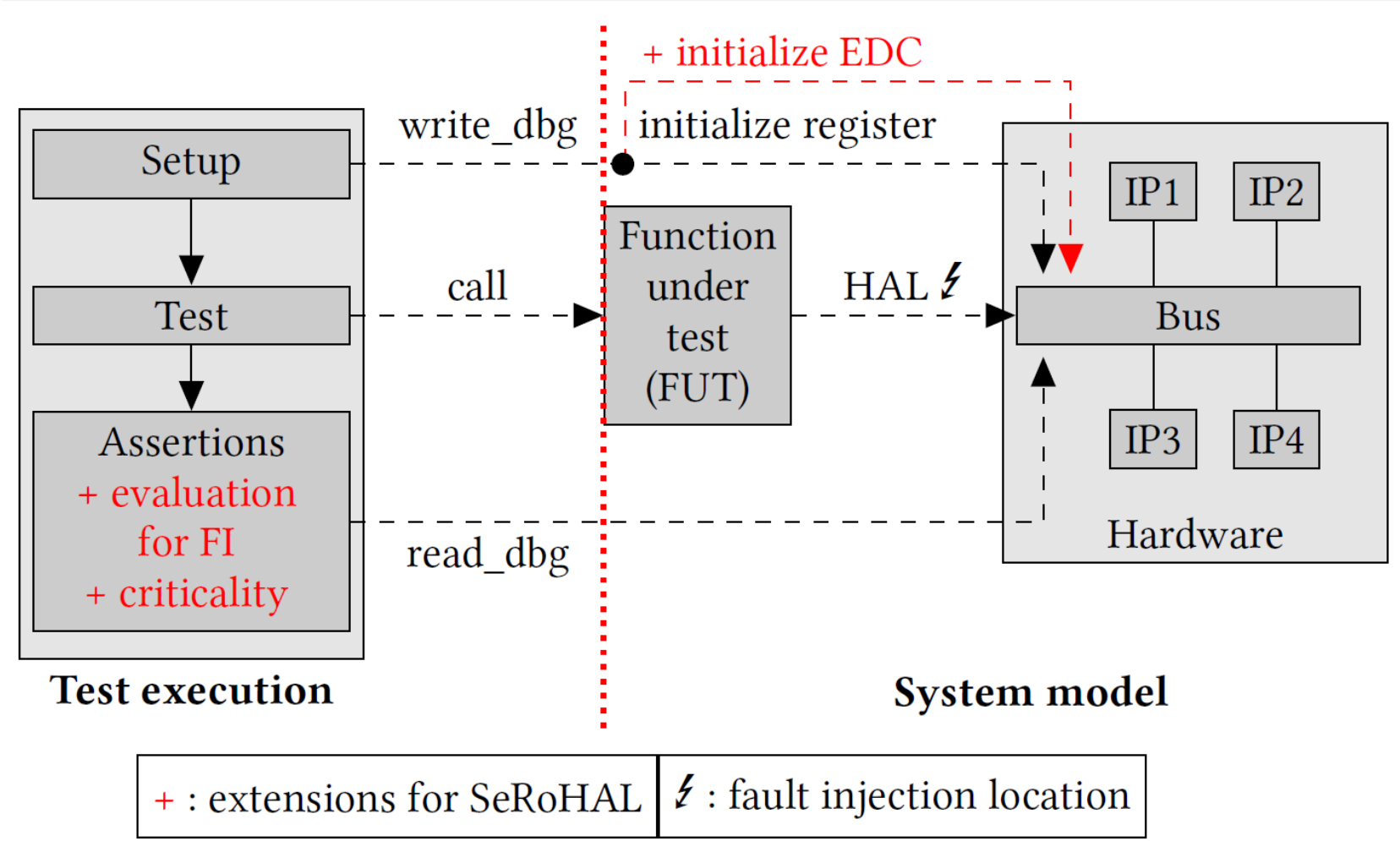
Handled errors

Protection mechanism	Protected bits					Detectable bit flip combinations
	rw	reserved	ro	wo	h	
Irrelevant bit mask GET		✓		✓		-
Irrelevant bit mask SET		✓				-
Readtwice	✓		✓			all
Readback	✓					all
Parity	✓					$n(1 \to 0) + n(0 \to 1) = 2k + 1$ with $k \in \mathbb{N}_0$
Checksum	✓					$n(1 \to 0) \neq n(0 \to 1)$
Word duplication	✓					all

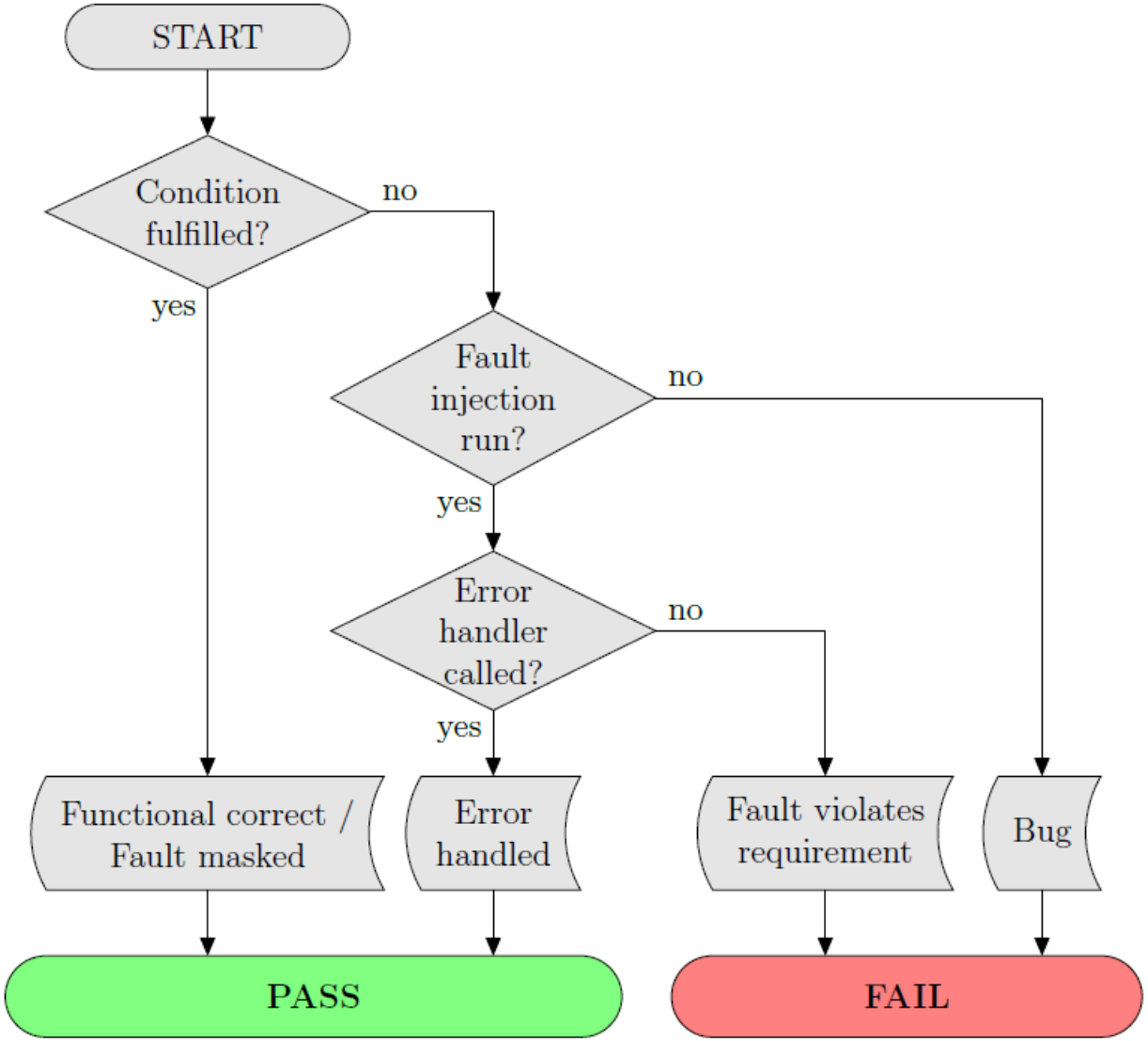
Handled error types

Protection mechanism	Bus error		Register error	
	during LD	during ST	transient	permanent
Irrelevant bit mask GET	✓	✓	✓	✓
Irrelevant bit mask SET	P	P	P	P
Readtwice	✓			
Readback		✓		next ST
Parity	✓	next LD	next LD	next LD
Checksum	✓	next LD	next LD	next LD
Word duplication	✓	next LD	next LD	next LD

Fault injection setup



Assertion classification



Generated HALs

	Unprotected	Mask irrelevant bits	Readtwice & Readback	EDCs only			EDCs & readback			Selective		
				Parity	Checksum	Word duplication	Parity	Checksum	Word duplication	Word duplication & readback for SIL 1-4	Word duplication & readback for SIL 3-4	mixed
Mask irrelevant bits		X	X	X	X	X	X	X	X	X	X	X
Readtwice			X									1
Parity				X			X					2,3
Checksum					X			X				
Word duplication						X			X	1-4	3-4	4
Readback			X				X	X	X	1-4	3-4	1,3,4

Overhead – ROM

- Include header files with protected HALs
- Cross-compile for target