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

Petra R. Kleeberger, Juana Rivera, Daniel Mueller-Gritschneider, 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:**
- Lockstep
- ECC for memory

**SW-based mechanisms inserted by compilers:**
- Instruction duplication
- Variable duplication
- Control-flow checking

Diagram:
- Memory
- Caches
- CPU
- Peripheral 1
- Peripheral 2
- Peripheral 3
Error Detection and Handling

Mechanisms traditionally added manually by designers:
- Integrity checks
- Checksums
- Readback
- Readtwice

- Error prone
- High design effort

HW-based mechanisms:
- Lockstep
- ECC for memory

SW-based mechanisms inserted by compilers:
- Instruction duplication
- Variable duplication
- Control-flow checking

Peripheral 1
Peripheral 2
Peripheral 3

Memory
Caches
CPU

Bus
Error Detection and Handling

Mechanisms traditionally added manually by designers:
- Integrity checks
- Checksums
- Readback
- Readtwice

- Error prone
- High design effort

HW-based mechanisms:
- Lockstep
- ECC for memory

SW-based mechanisms inserted by compilers:
- Instruction duplication
- Variable duplication
- Control-flow checking

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>
<thead>
<tr>
<th>Application software</th>
<th>InitTimer()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register-specific HAL</td>
<td>TIMER_CTRL_SET(value)</td>
</tr>
<tr>
<td>Low-level HAL</td>
<td>REG_WRITE32(addr, value)</td>
</tr>
<tr>
<td>Hardware</td>
<td>Register CTRL in Timer</td>
</tr>
<tr>
<td></td>
<td>31 ... 3 2 1 0</td>
</tr>
<tr>
<td></td>
<td>RESERVED F2 F1 F0</td>
</tr>
<tr>
<td></td>
<td>r w rwh rw</td>
</tr>
</tbody>
</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

<table>
<thead>
<tr>
<th>Application software</th>
<th>InitTimer()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register-specific HAL</td>
<td>TIMER_CTRL_SET(value)</td>
</tr>
<tr>
<td>Low-level HAL</td>
<td>REG_WRITE32(addr, value)</td>
</tr>
<tr>
<td>Hardware</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register CTRL in Timer</th>
<th>31</th>
<th>...</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESERVED</td>
<td>F2</td>
<td>F1</td>
<td>F0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r</td>
<td>w</td>
<td>rwh</td>
<td>rw</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**SeRoHAL:** add error detection mechanisms  
Wrong values loaded/stored  
Errors occur  
No/less errors
Register-specific HAL Generation

Code templates

Hardware description

HAL Generator

Header files

void TIMER_CTRL__SET( value ) {
    addr = 0x10000004;
    REG_WRITE32(addr, value);
}

uint32_t TIMER_CTRL__GET( ) {
    addr = 0x10000004;
    value = REG_READ32(addr);
    return value;
}
Register-specific HAL Generation

Code templates

```c
void ${IP}_${REG}_SET( value ) {
    addr = ${REG_ADDR};
    REG_WRITE32(addr, value);
}

uint32_t ${IP}_${REG}_GET( ) {
    addr = ${REG_ADDR};
    value = REG_READ32(addr);
    return value;
}
```

Header files

```c
void TIMER_CTRL_SET( value ) {
    addr = 0x10000004;
    REG_WRITE32(addr, value);
}

uint32_t TIMER_CTRL_GET( ) {
    addr = 0x10000004;
    value = REG_READ32(addr);
    return value;
}
```

Hardware description

HAL Generator

${…}: placeholder
Register-specific HAL Generation

Code templates

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

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

${…}: placeholder

Header files

void TIMER_CTRL__SET( value ) {
    addr = 0x10000004;
    REG_WRITE32(addr, value);
}

uint32_t TIMER_CTRL__GET( ) {
    addr = 0x10000004;
    value = REG_READ32(addr);
    return value;
}

Hardware description

IP: TIMER
    Address: 0x10000000
    ... Register: CTRL
        Offset: 0x4
        Field: F0, rw, 1bit, offset 0
        Field: F1, rwh, 1bit, offset 1
        ...
Register-specific HAL Generation

Code templates

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

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

Header files

```c
void TIMER_CTRL__SET( value ) {
    addr = 0x10000004;
    REG_WRITE32(addr, value);
}

uint32_t TIMER_CTRL__GET( ) {
    addr = 0x10000004;
    value = REG_READ32(addr);
    return value;
}
```

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

```c
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;
}
```

<table>
<thead>
<tr>
<th>31</th>
<th>...</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>w</td>
<td>rwh</td>
<td>rw</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Read as zero**

```
0x00000003
```
## Mask Irrelevant Bits

```c
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;
}
```

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>...</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RESERVED</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>F2</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>F1</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>F0</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>r</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>w</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>rwh</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>rw</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Read as zero

- **0x00000003**
- **0x10000003**

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

Bus error

- **0x10000003**
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;
}

START

v2 = REG_READ32(addr);
v1 = value & ${RT_MASK};
v2 = v2 & ${RT_MASK};

true

false

error_handler( addr );

END

Mask in both loaded values bits that are:
• Volatile
• Write-only
• Reserved
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;
}

Mask in stored and loaded value bits that are:
- Volatile
- Write-only
- Reserved
- Read-only

true

false

v1 == v2

error_handler( addr );

START

END
Error Detection Codes (EDCs) – Word Duplication

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

value = value & ${EDC_MASK};
edc = value;
edc_addr = ${EDC_ADDR};
REG_WRITE32(edc_addr, edc);

0xFFFFF80000000000
write

0x00000003
read

31 ... 3 2 1 0
0 ... 0 1 1 1
r w rwh rw

START

END
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

<table>
<thead>
<tr>
<th>Protected registers</th>
<th>EDC memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>REG_ADDR</td>
<td></td>
</tr>
<tr>
<td>0x10000000</td>
<td>Reg1</td>
</tr>
<tr>
<td>0x10000004</td>
<td>Reg2</td>
</tr>
<tr>
<td>0x10000008</td>
<td>Reg3</td>
</tr>
</tbody>
</table>

Word duplication (WD):

<table>
<thead>
<tr>
<th>EDC_ADDR</th>
<th>WD1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x40000000</td>
<td></td>
</tr>
<tr>
<td>0x40000004</td>
<td></td>
</tr>
<tr>
<td>0x40000008</td>
<td></td>
</tr>
</tbody>
</table>

Checksum (CS):  

Parity (P):
## Error Detection Codes – Memory Layout

<table>
<thead>
<tr>
<th>Protected registers</th>
<th>EDC memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>word duplication (WD):</td>
<td></td>
</tr>
<tr>
<td>EDC_ADDR:</td>
<td></td>
</tr>
<tr>
<td>0x40000000 WD1</td>
<td></td>
</tr>
<tr>
<td>0x40000004 WD2</td>
<td></td>
</tr>
<tr>
<td>0x40000008 WD3</td>
<td></td>
</tr>
<tr>
<td>checksum (CS):</td>
<td></td>
</tr>
<tr>
<td>EDC_ADDR:</td>
<td></td>
</tr>
<tr>
<td>0x40000000</td>
<td>CS3</td>
</tr>
<tr>
<td><em>EDC_START_BIT</em>: 31 … 18 12 6 0</td>
<td></td>
</tr>
<tr>
<td>parity (P):</td>
<td></td>
</tr>
<tr>
<td>EDC_ADDR:</td>
<td></td>
</tr>
<tr>
<td>0x40000000</td>
<td>P3</td>
</tr>
<tr>
<td><em>EDC_START_BIT</em>: 31 … 3 2 1 0</td>
<td></td>
</tr>
</tbody>
</table>

### Protected registers
- REG_ADDR
  - 0x10000000: Reg1
  - 0x10000004: Reg2
  - 0x10000008: Reg3
## Error Detection Codes (EDCs)

<table>
<thead>
<tr>
<th></th>
<th>Word duplication</th>
<th>Checksum</th>
<th>Parity</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDC storage overhead</td>
<td>High</td>
<td>Medium</td>
<td>Low</td>
</tr>
<tr>
<td>Needs EDC extraction after load</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>code size overhead</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Needs read-modify-write operation to store EDC</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>code size and performance overhead</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Error detection capability</td>
<td>High</td>
<td>Medium</td>
<td>Low</td>
</tr>
</tbody>
</table>
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

Setup
- Software state:
  global variables, …
- Hardware model:
  registers, …

Run simulation
…
TIMER_CTRL_GET( );
TIMER_CTRL_SET( );
…

Evaluate
…
ASSERT(var1==0);
ASSERT(var2==1);
…
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

Setup
• Software state: global variables, …
• Hardware model: registers, …

Run simulation
…
TIMER_CTRL_GET( );
TIMER_CTRL_SET( );
…

Evaluate
…
ASSERT(var1==0, QM);
ASSERT(var2==1, SIL4);
…
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:

<table>
<thead>
<tr>
<th>Function</th>
<th>Criticality</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIMER_CTRL_GET</td>
<td>SIL4</td>
</tr>
<tr>
<td>TIMER_CTRL_SET</td>
<td>QM</td>
</tr>
</tbody>
</table>

Setup
- Software state: global variables, …
- Hardware model: registers, …

Run simulation
- TIMER_CTRL_GET( );
- TIMER_CTRL_SET( );

Evaluate
- ASSERT(var1==0, QM);
- ASSERT(var2==1, SIL4);
- ...

29
## Selective Protection

<table>
<thead>
<tr>
<th></th>
<th>SIL4</th>
<th>QM</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIMER_CTRL_GET</td>
<td>word duplication</td>
<td>no protection</td>
</tr>
<tr>
<td>TIMER_CTRL_SET</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Protection policy:

<table>
<thead>
<tr>
<th>SIL Level</th>
<th>GET</th>
<th>SET</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIL 4</td>
<td>word duplication</td>
<td>readback</td>
</tr>
<tr>
<td>SIL 3</td>
<td>checksum</td>
<td>readback</td>
</tr>
<tr>
<td>SIL 2</td>
<td>parity</td>
<td>readback</td>
</tr>
<tr>
<td>SIL 1</td>
<td>parity</td>
<td>no protection</td>
</tr>
<tr>
<td>QM</td>
<td>no protection</td>
<td>no protection</td>
</tr>
</tbody>
</table>
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 overhead

QM
SIL 1
SIL 2
SIL 3
SIL 4
Robustness

Number of failed assertions

3% reduction, nearly no overhead

76% reduction, but high overhead

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

36% reduction
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>
    </fields>
  </register>
</registers>
<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>

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Bit Offset</th>
<th>Bit Width</th>
<th>Access</th>
<th>Volatile</th>
</tr>
</thead>
<tbody>
<tr>
<td>EN</td>
<td>Timer enable flag</td>
<td>1</td>
<td>1</td>
<td>write-only</td>
<td></td>
</tr>
<tr>
<td>INT_STAT</td>
<td>Interrupt status flag</td>
<td>2</td>
<td>1</td>
<td>read/write</td>
<td>true</td>
</tr>
</tbody>
</table>

```
31   ...   3  2  1  0
RESERVED INT_STAT EN INT_EN
r   rwh   w   rw
```
## Masks

<table>
<thead>
<tr>
<th>31</th>
<th>...</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>F2</strong></td>
<td><strong>F1</strong></td>
<td><strong>F0</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ro</td>
<td>wo</td>
<td>rwh</td>
<td>rw</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**RESERVED**

<table>
<thead>
<tr>
<th>31</th>
<th>...</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>F2</strong></td>
<td><strong>F1</strong></td>
<td><strong>F0</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ro</td>
<td>wo</td>
<td>rwh</td>
<td>rw</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**RESERVED MASK** = 0x00000007 = 0b

<table>
<thead>
<tr>
<th>31</th>
<th>...</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>000...000</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**RO_MASK** = 0x00000007 = 0b

<table>
<thead>
<tr>
<th>31</th>
<th>...</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>000...000</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**WO_MASK** = 0xFFFFFFFFFB = 0b

<table>
<thead>
<tr>
<th>31</th>
<th>...</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>111...111</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**H_MASK** = 0xFFFFFFFFFD = 0b

<table>
<thead>
<tr>
<th>31</th>
<th>...</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>111...111</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

```
value = value & ${EDC_MASK};
edc1 = parity(value);
edc_addr = ${EDC_ADDR};
edc2 = REG_READ32(edc_addr);
edc2 = (edc2 >> ${EDC_START_BIT}) & 0x1;
```

```
edc1 == edc2
```

START
true
false

```
error_handler( addr );
```

END
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

```c
value = value & ${EDC_MASK};
edc = parity(value);

edc_addr = ${EDC_ADDR};
edc_word = REG_READ32(edc_addr);
edc_word &= ~(0x1 << ${EDC_START_BIT});
edc_word |= edc << ${EDC_START_BIT};
REG_WRITE32(edc_addr, edc_word);
```
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

\[
\begin{align*}
\text{tmp} &= \text{value} \& \$\{\text{EDC\_PROTECTION\_MASK}\}; \\
\text{edc\_addr} &= \$\{\text{EDC\_ADDR}\}; \\
\text{ld\_edc} &= \text{REGISTER\_READ32}(\text{edc\_addr}); \\
\text{edc} &= \text{wd}(\text{tmp}); \\
\text{edc\_mask} &= 0x3F; \\
\text{edc\_bit} &= \$\{\text{EDC\_START\_BIT}\}; \\
\text{ld\_edc} &= (\text{ld\_edc} \gg \text{edc\_bit}) \& \text{edc\_mask}; \\
\text{edc} &= \text{ld\_edc}; \\
\text{error\_handler}(\text{addr}); \\
\end{align*}
\]
EDC memory layout

<table>
<thead>
<tr>
<th>EDC_ADDR</th>
<th>WD1</th>
<th>WD2</th>
<th>P9</th>
<th>P8</th>
<th>CS7</th>
<th>CS6</th>
<th>CS5</th>
<th>CS4</th>
<th>CS3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x400000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x40000004</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x40000008</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

EDC_START_BIT | 31 | 30 | 24 | 18 | 12 | 6  | 0  |

51
Overheads induced by protection mechanisms

<table>
<thead>
<tr>
<th>Protection mechanism</th>
<th>RAM overhead per l-bit register [bits]</th>
<th>Additional HW accesses per load access</th>
<th>Additional HW accesses per store access</th>
</tr>
</thead>
<tbody>
<tr>
<td>Irrelevant bit mask</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Readback</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Readtwice</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Parity</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Checksum</td>
<td>$\lceil \log_2(l + 1) \rceil$</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Word duplication</td>
<td>$l$</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
## Handled errors

<table>
<thead>
<tr>
<th>Protection mechanism</th>
<th>Protected bits</th>
<th>Detectable bit flip combinations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>rw reserved ro wo h</td>
<td></td>
</tr>
<tr>
<td>Irrelevant bit mask GET</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>Irrelevant bit mask SET</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>Readtwice</td>
<td>✓</td>
<td>all</td>
</tr>
<tr>
<td>Readback</td>
<td>✓</td>
<td>all</td>
</tr>
<tr>
<td>Parity</td>
<td>✓</td>
<td>(n(1 \rightarrow 0) + n(0 \rightarrow 1) = 2k + 1) with (k \in \mathbb{N}_0)</td>
</tr>
<tr>
<td>Checksum</td>
<td>✓</td>
<td>(n(1 \rightarrow 0) \neq n(0 \rightarrow 1)) all</td>
</tr>
<tr>
<td>Word duplication</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>
## Handled error types

<table>
<thead>
<tr>
<th>Protection mechanism</th>
<th>Bus error during LD</th>
<th>Bus error during ST</th>
<th>Register error transient</th>
<th>Register error permanent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Irrelevant bit mask GET</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Irrelevant bit mask SET</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>Readtwice</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Readback</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parity</td>
<td>✓</td>
<td>next LD</td>
<td>next LD</td>
<td>next ST</td>
</tr>
<tr>
<td>Checksum</td>
<td>✓</td>
<td>next LD</td>
<td>next LD</td>
<td>next LD</td>
</tr>
<tr>
<td>Word duplication</td>
<td>✓</td>
<td>next LD</td>
<td>next LD</td>
<td>next LD</td>
</tr>
</tbody>
</table>
Fault injection setup

Test execution

- Setup
- Test
- Assertions
  - + evaluation for FI
  - + criticality

System model

- Function under test (FUT)
- HAL
- Hardware
  - IP1
  - IP2
  - IP3
  - IP4

write_dbg
initialize register
initialize EDC

+ : extensions for SeRoHAL
/: fault injection location
Assertion classification

START

Condition fulfilled?

yes

Fault injection run?

yes

Error handler called?

yes

Functional correct / Fault masked

PASS

no

no

no

Error handled

FAIL

Fault violates requirement

Bug
## Generated HALs

<table>
<thead>
<tr>
<th></th>
<th>Unprotected</th>
<th>Mask irrelevant bits</th>
<th>Readtwice &amp; Readback</th>
<th>EDCs only</th>
<th>Parity</th>
<th>Checksum</th>
<th>Word duplication</th>
<th>EDCs &amp; readback</th>
<th>Parity</th>
<th>Checksum</th>
<th>Word duplication</th>
<th>Selective</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Mask irrelevant bits</strong></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td><strong>Readtwice</strong></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td><strong>Parity</strong></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>2,3</td>
</tr>
<tr>
<td><strong>Checksum</strong></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Word duplication</strong></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>1-4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Readback</strong></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>1-4</td>
<td>3-4</td>
<td></td>
<td>1,3,4</td>
</tr>
</tbody>
</table>
Overhead – ROM

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