

Learning the Portable Stimulus Standard

Tom Fitzpatrick

CEO Big Fish EDA Consulting

On behalf of the Accellera Systems Initiative

and the Accellera Portable Stimulus Working Group



Registers

 Session 14

What You Will Learn

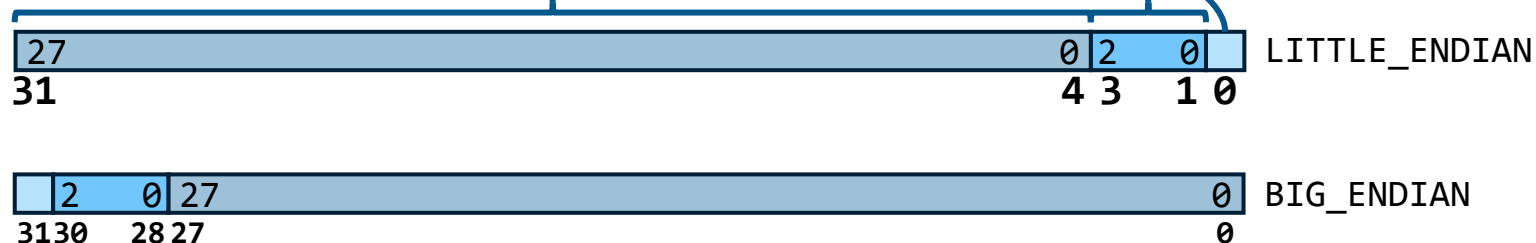
- ▶ Packed structs to define multi-field registers
- ▶ Modeling individual registers
- ▶ Grouping registers
- ▶ Calculating register offsets
- ▶ Assigning register groups to address regions

Modeling Registers: Packed Structs

- ▶ A *packed struct* ensures contiguous field alignment
- ▶ Include numeric, boolean, typed enum, packed structs and arrays of these

```
enum endianness_e {LITTLE_ENDIAN, BIG_ENDIAN};  
struct packed_s <endianness_e e = LITTLE_ENDIAN> {};
```

```
struct ctrl_s : packed_s<> {  
    bit go_done;  
    bit[3] mode;  
    bit[28] reserved;  
}
```



Modeling Registers

- ▶ The `reg_c` component is the base type for specifying registers
- ▶ **pure** components only contain realization-level functionality

```
package addr_reg_pkg {  
  enum reg_access {READWRITE, READONLY, WRITEONLY};  
  
  pure component reg_c < type R, reg_access ACC = READWRITE, int SZ = (8*sizeof_s<R>::nbytes)> {  
    target function R read();  
    target function void write(R r);  
    target function bit[SZ] read_val();  
    target function void write_val(bit[SZ] r);  
    target function void write_masked(R mask, R val);  
    target function void write_val_masked(bit[SZ] mask, bit[SZ] val);  
    target function void write_field(string name, bit[SZ] val);  
    target function void write_fields(list<string> names, list<bit[SZ]> vals);  
  }  
  ...  
}
```

Packed struct or bit vector

SZ = #bits in the register

$\text{REG_VAL}(\text{new}) = (\text{REG_VAL}(\text{current}) \& \sim \text{mask}) \mid (\text{val} \& \text{mask})$

::: Modeling Registers: Register Groups

- ▶ The `reg_group_c` component aggregates registers and other groups
- ▶ A register group may be associated with an address region

```
package addr_reg_pkg {  
  ...  
  struct node_s {  
    string name;  
    int index;  
  };  
  
  pure component reg_group_c {  
    pure function bit[64] get_offset_of_instance(string name);  
    pure function bit[64] get_offset_of_instance_array(string name, int index);  
    pure function bit[64] get_offset_of_path(list<node_s> path);  
    solve function void set_handle(addr_handle_t addr);  
  };  
}
```

::: Modeling Registers: Layout

- ▶ The **reg_c** component is parameterized by the register layout
 - ▶ bit vector or packed struct

```
package my_reg_pkg {  
  import addr_reg_pkg::*;  
  
  struct ctrl_s : packed_s<> {  
    bit go_done;  
    bit[3] mode;  
    bit[28] reserved;  
  }  
}
```

```
pure component src_reg_c : reg_c<bit[32]>{}  
pure component dst_reg_c : reg_c<bit[32]>{}  
pure component size_reg_c : reg_c<bit[32]>{}  
pure component ctrl_reg_c : reg_c<ctrl_s>{}
```

src_reg
dst_reg
size_reg
ctrl_reg

Register Offset of Instance

- ▶ Each register group must include user-defined methods to calculate offsets
- ▶ For individual registers or groups, use `get_offset_of_instance()`

```
package my_reg_pkg {  
  import addr_reg_pkg::*;
```

```
  pure component dma_chan_reg_grp : reg_group_c {  
    src_reg_c  src_reg;  
    dst_reg_c  dst_reg;  
    size_reg_c size_reg;  
    ctrl_reg_c ctrl_reg;
```

src_reg
dst_reg
size_reg
ctrl_reg

```
  function bit[64] get_offset_of_instance(string name) {  
    match(name) {  
      ["src_reg"] : return 0x0;  
      ["dst_reg"] : return 0x4;  
      ["size_reg"] : return 0x8;  
      ["ctrl_reg"] : return 0xc;  
      default : return -1;  
    }  
  }
```

ctrl_reg	0C
size_reg	08
dst_reg	04
src_reg	00

```
  grp.dst_reg.offset = grp.offset + dma_chan_reg_grp.get_offset_of_instance("dst_reg")
```

Register Offset of Instance Array

- ▶ Each register group must include user-defined methods to calculate offsets
- ▶ For arrays or registers or groups, use `get_offset_of_instance_array()`

```
package my_reg_pkg {  
  import addr_reg_pkg::*;  
  
  pure component dma_reg_grp : reg_group_c {  
    dma_chan_reg_grp chan_regs[4];  
  
    function bit[64] get_offset_of_instance(string name) {  
      return -1; // no specific instances, arrays only  
    }  
    function bit[64] get_offset_of_instance_array(string name, int index) {  
      match(name) {  
        ["chan_regs"] : return (0x10 * index);  
        default : return -1;  
      }  
    }  
  }  
}
```

ctrl_reg	3C
size_reg	38
dst_reg	34
src_reg	30
ctrl_reg	2C
size_reg	28
dst_reg	24
src_reg	20
ctrl_reg	1C
size_reg	18
dst_reg	14
src_reg	10
ctrl_reg	0C
size_reg	08
dst_reg	04
src_reg	00

`grp.chan_regs[N].dst_reg.offset = grp.offset + get_offset_of_instance_array("chan_regs", N) + chan_regs[N].get_offset_of_instance("dst_reg")`

Register Offset of Path

- ▶ `get_offset_of_path()` method is used if it is defined
- ▶ Uses a list of `node_s` ({string name; int index;}) to identify register

```
package my_reg_pkg {  
  import addr_reg_pkg::*;  
  
  pure component dma_reg_grp : reg_group_c {  
    dma_chan_reg_grp chan_regs[4];  
    int chan;  
  
    function bit[64] get_offset_of_path(list<node_s> path) {  
      if (path[0].name == "chan_regs") {  
        chan = 0x10 * path[0].index;  
        if (path[1].name == "src_reg") {return chan + 0x0; };  
        else if (path[1].name == "dst_reg") {return chan + 0x4; };  
        else if (path[1].name == "size_reg") {return chan + 0x8; };  
        else if (path[1].name == "ctrl_reg") {return chan + 0xc; };  
        else {return -1;};  
      }  
      else {return -1;};  
    }  
  }  
}
```

`grp.chan_regs[N].dst_reg.offset = grp.offset + get_offset_of_path({path[0],path[1]}, {"chan_regs",N}, {"dst_reg",0})`

ctrl_reg	3C
size_reg	38
dst_reg	34
src_reg	30
ctrl_reg	2C
size_reg	28
dst_reg	24
src_reg	20
ctrl_reg	1C
size_reg	18
dst_reg	14
src_reg	10
ctrl_reg	0C
size_reg	08
dst_reg	04
src_reg	00

Example

- Use `set_handle()` to associate a `reg_group_c` with an address region

```
component dma_c {  
  import addr_reg_pkg::*;  
  import ex14_reg_pkg::*;  
  transparent_addr_space_c<> aspace;  
  transparent_addr_region_s<> regs;  
  dma_reg_grp dma_regs;  
  
  exec init_up {  
    regs.addr = 0x00001000 ;  
    regs.size = 0x00002000;  
    regs_h = aspace.add_nonallocatable_region(regs);  
    dma_regs.set_handle(regs_h);  
  }  
}
```

aspace		
regs		
ctrl_reg	3C	
size_reg	38	
dst_reg	34	
src_reg	30	
ctrl_reg	2C	
size_reg	28	
dst_reg	24	
src_reg	20	
ctrl_reg	1C	
size_reg	18	
dst_reg	14	
src_reg	10	
ctrl_reg	0C	
size_reg	08	
dst_reg	04	
src_reg	00	

ctrl_reg	3C
size_reg	38
dst_reg	34
src_reg	30
ctrl_reg	2C
size_reg	28
dst_reg	24
src_reg	20
ctrl_reg	1C
size_reg	18
dst_reg	14
src_reg	10
ctrl_reg	0C
size_reg	08
dst_reg	04
src_reg	00

Example

- ▶ Register references converted to address handles based on calculated offsets
- ▶ Register accesses converted to built-in primitive access API calls

```
action xfer_a {  
  input mbuf_b ibuf;  
  output mbuf_b obuf;  
  lock chan_r chan;
```

```
  exec body {  
    src_hndl = make_handle_from_claim(ibuf.mClaim);  
    dst_hndl = make_handle_from_claim(obuf.mClaim);  
    comp.dma_regs.chan_regs[chan.instance_id].size_reg.write(ibuf.size);  
    comp.dma_regs.chan_regs[chan.instance_id].src_reg.write(addr_value(src_hndl));  
    comp.dma_regs.chan_regs[chan.instance_id].dst_reg.write(addr_value(dst_hndl));  
    comp.dma_regs.chan_regs[chan.instance_id].ctrl_reg.write_field("go_done", 0x1);  
    while (comp.dma_regs.chan_regs[chan.instance_id].ctrl_reg.read().go_done == 1) {  
      message(NONE, "Waiting for DMA channel %d to complete", chan.instance_id);  
    }  
  }  
}
```

```
comp.dma_regs.chan_regs[chan.instance_id].size_reg.write(ibuf.size);
```

```
wr_h = make_handle_from_handle(regs_h, offset);  
write32(wr_h, ibuf.size);
```

aspace	
regs	
ctrl_reg	3C
size_reg	38
dst_reg	34
src_reg	30
ctrl_reg	2C
size_reg	28
dst_reg	24
src_reg	20
ctrl_reg	1C
size_reg	18
dst_reg	14
src_reg	10
ctrl_reg	0C
size_reg	08
dst_reg	04
src_reg	00

Can be mapped to the appropriate executor



Thank You

