csco_addrmap eVC Overview

Joseph H. Zhang

July 2006

Agenda

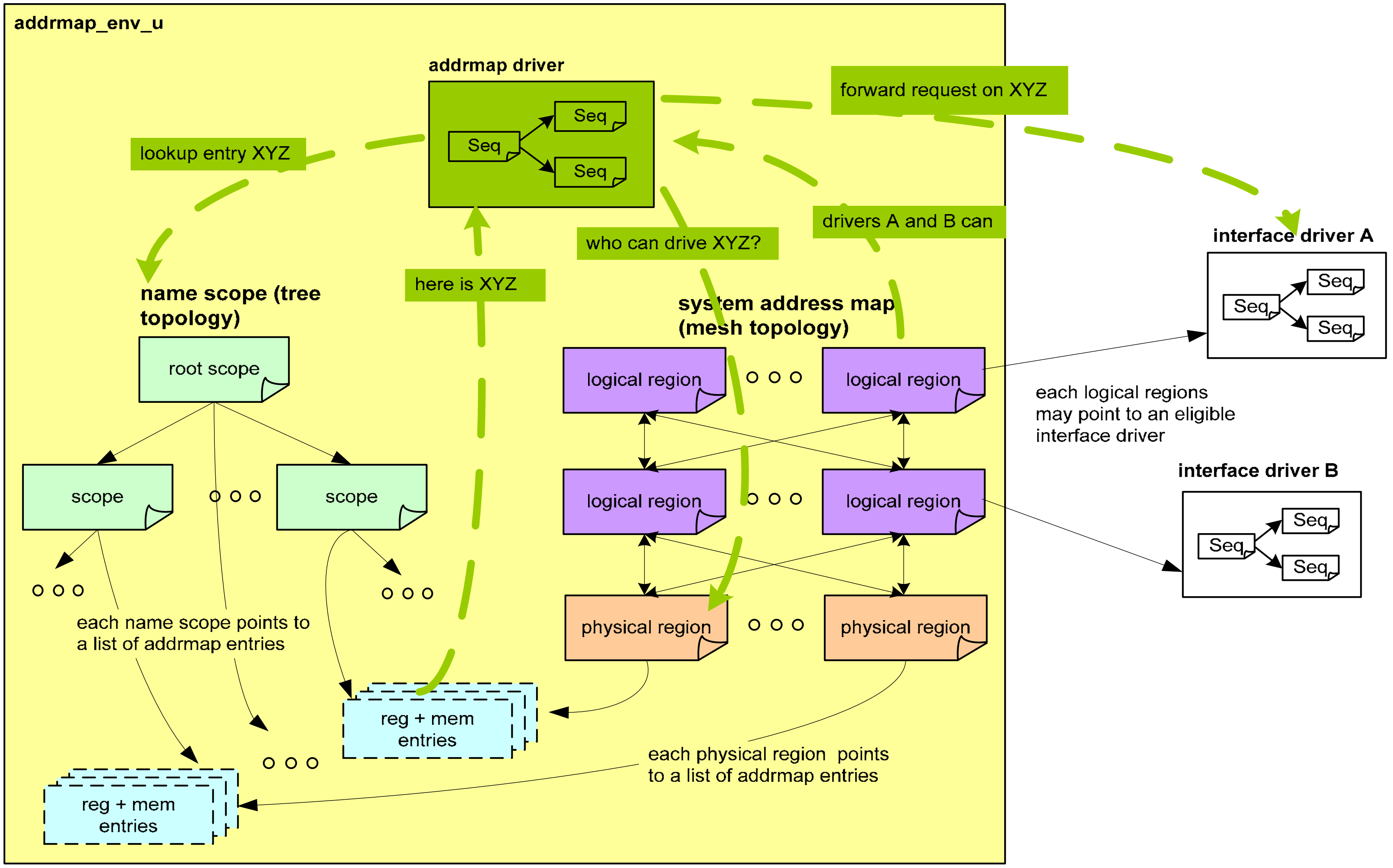
- Motivation
- Theory of Operation
- Usage Model
- Legacy Support

Motivation

- Efficient register map modeling.
  - Easily supports 25K register and memory entries.
  - Current ASIC register space is approaching 10K entries.
- Support for vertical reuse.
  - Interface independent CPU transaction interface.
  - Hierarchical name scopes.
- Enables early testing of DUT.
  - Automatic shadow value tracking and scoreboarding.
  - Predefined sequence library.

    REG_POR, WALK0/WALK1, INTERMIX, etc…
  - ** Predefined register access functional coverage.

Theory of Operation - Architecture



Theory of Operation – Register Modeling

- Efficient memory usage:
  - All register and field instances are of the same exact type – no ‘when’ sub-typing.
  - Register and field instances are uniquely identified via string names.
- Hierarchical name space:
  - Each sequence/transaction has a user configurable scope to ensure vertical reuse.
- Three types of address map entries:
  - Register : list of fields.
  - Symbolic Memory : multiple rows of fields, where all rows have the same list of fields.
  - Byte Memory : No logical fields, purely byte based memory with configurable Endianness.

Theory of Operation - Tight Integration with REGTOOL (Blueprint)

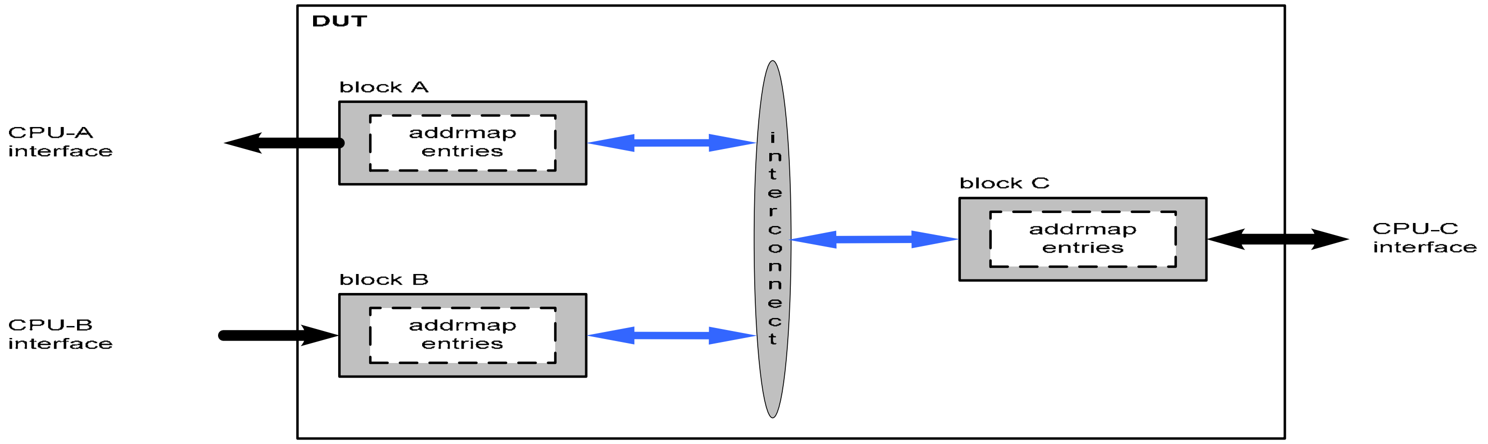
- **Full support for all Blueprint components.**
- **Dynamic XML parser reduces HVL code base.**
- **Extensible architecture to support any register automation tool.**

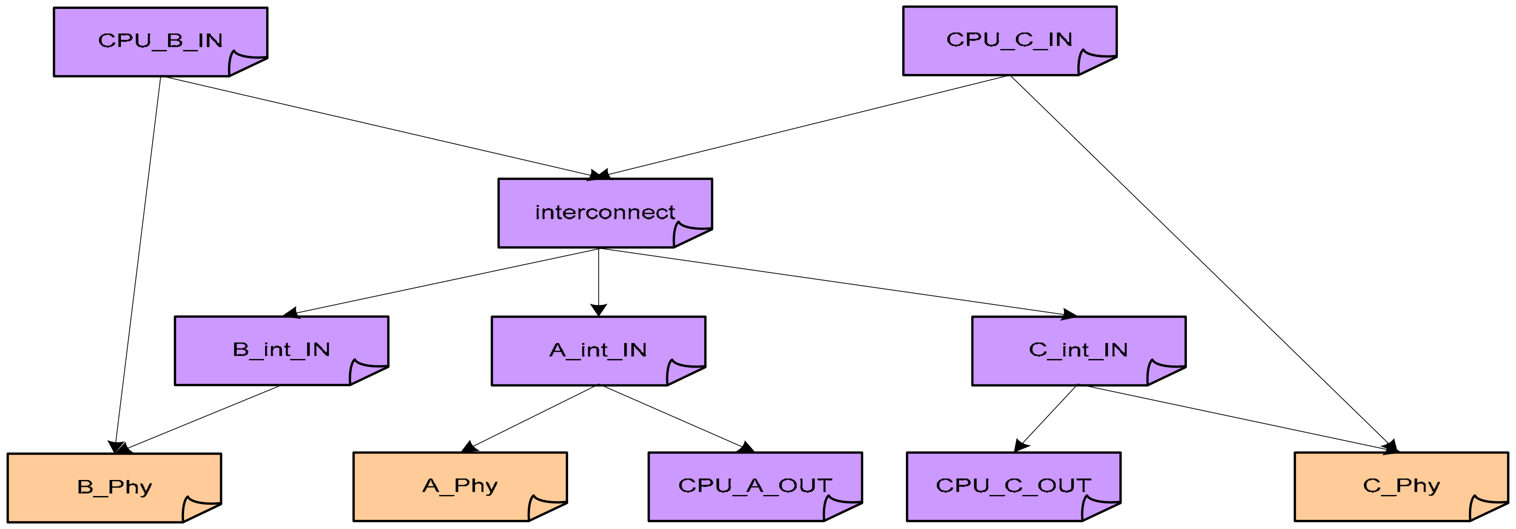
Blueprint	csco_addrmap
Field	csco_addrmap_field_s
Register	csco_addrmap_reg_entry_s
Register Array	csco_addrmap_symb_mem_entry_s
Register File	csco_addrmap_name_scope_s
Address Map	csco_addrmap_name_scope_s csco_addrmap_physical_region_s csco_addrmap_logical_region_s
Side-Effects	csco_addrmap_side_effect_t
Access Modes	csco_addrmap_sw_access_t

Theory of Operation – Address Map Modeling

- **Physical region:**
  - **A container for a group of physical entries each with a unique local offset.**
- **Logical region:**
  - **Contains one or more child physical or logical regions.**
  - **Amalgamate child regions which are accessed via the same interface.**
  - **Logical grouping of child regions for ease of manipulation.**
  - **Contains at most 1 pointer to a sequence driver that drives its interface.**
  - **Configurable/extensible algorithm for assigning offsets to child regions.**

Theory of Operation – Address Map Modeling



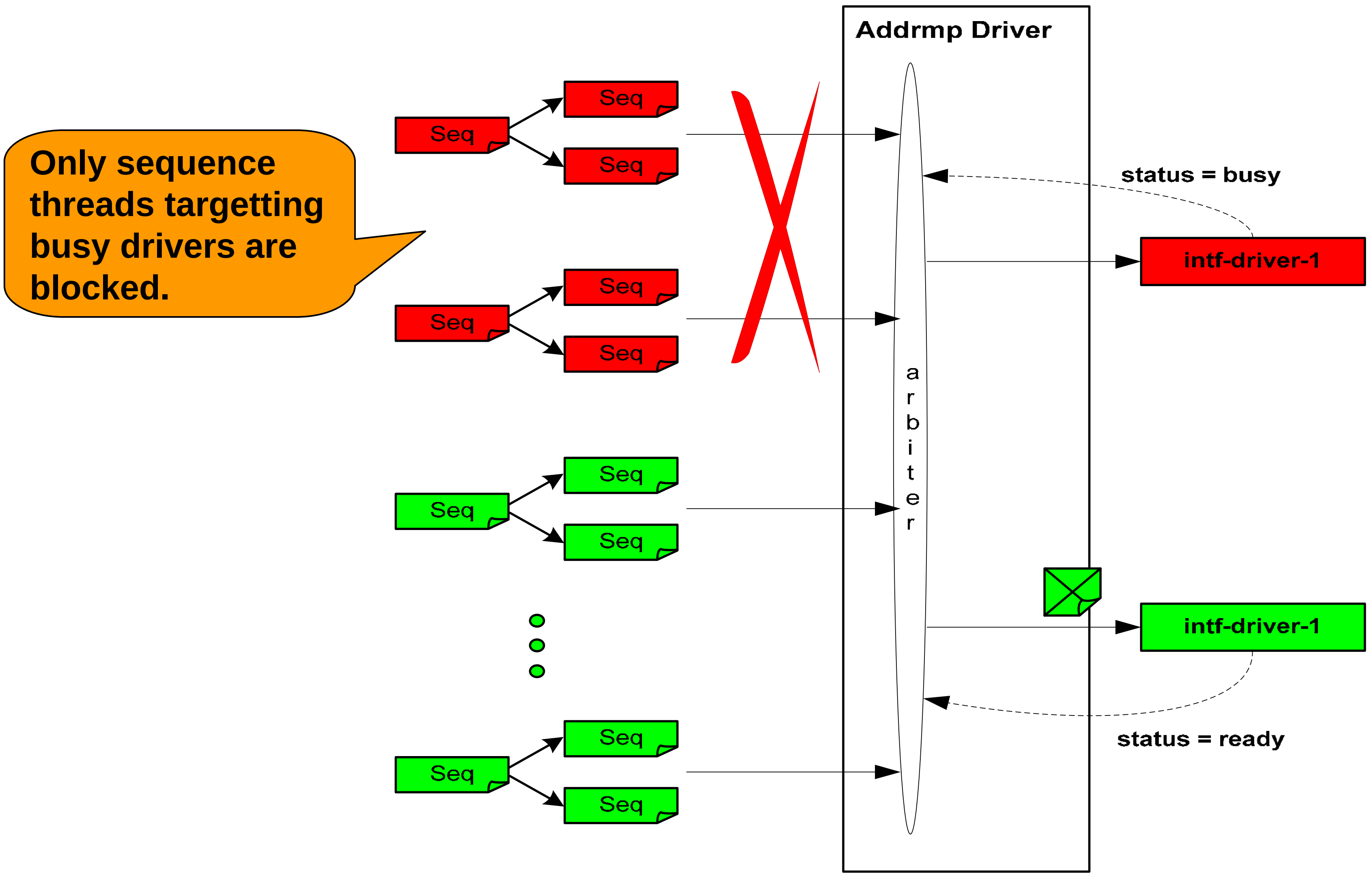


Theory of Operation - Sequence Interface

- Automatic/random transaction forwarding to source interface driver based on address map.
- Built-in segmentation engine (iterator) to map addrmap transaction to arbitrary interface alignments.
- Per-interface sequence backpressure – no driver head-of-line blocking.
- Two addressing schemes:
  - ENTRY: caller specifies entry/field names and values.
  - OFFSET: caller specifies target region and numeric offset.
- Special macro, ‘addrmap_do’, replaces Specman ‘do’ syntax.

Theory of Operation - Sequence Flow Control

Cisco.com

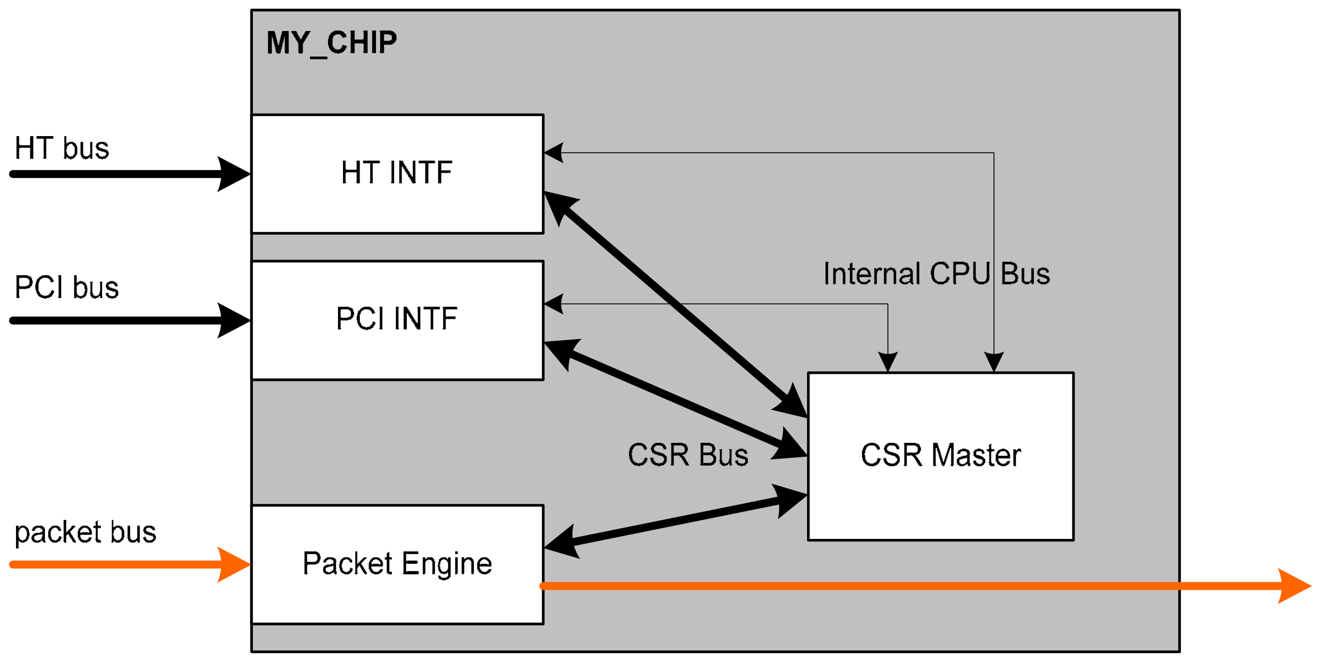


Posted versus Blocking

	resp_kind = POSTED	resp_kind = NON_POSTED
blocking_kind = NON_BLOCKING	*addrmap_do* returns in 0 time.	*addrmap_do* returns in 0 time.
blocking_kind = BLOCKING	*addrmap_do* returns once request has been transmitted over bus.	*addrmap_do* returns once response has been sent over request bus.

- **Uses linking mechanism defined in csco_trans package to detect transaction progress through system.**

Usage Model – Example DUT



Usage Model – Integrating Blueprint

Unit-Level

```
extend MY_CHIP csco_addrmap_env_u {
  post_generate() is also {
    scope.trim_all_except("/PKT_ENG/");
  };
};
```

PKT_ENG unit bench specific logic

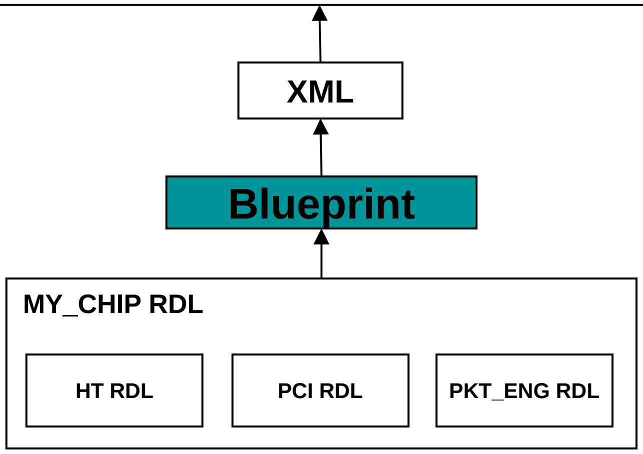
can be any regular expression

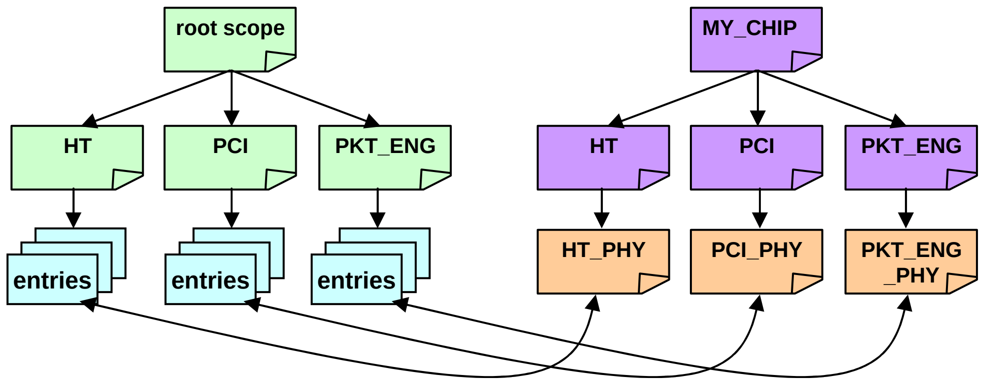
```
extend MY_CHIP csco_addrmap_env_u {
  !bp_region : csco_addrmap_logical_region_s;

  post_generate() is also {
    bp_region = scope.populate_from_xml_file("MY_CHIP.xml", NULL);
  };

};
```

<proj>_common_top.e file – same logic for all testbenches.





Usage Model – Integrating Blueprint

Cisco.com

Unit-Level

```
extend MY_CHIP csco_addrmap_env_u {
  connect_pointers() is also {
    var blk_region := bp_region.get_region_first_match("PKT_ENG");
    blk_region.set_eligible_driver(tb_ptr.csr_master.driver);
};
```

Add access to registers via CSR bus.

```
extend MY_CHIP csco_addrmap_env_u {
  post_generate() is also {
    scope.trim_all_except("/PKT_ENG/");
  };
};
```

PKT_ENG unit bench specific logic

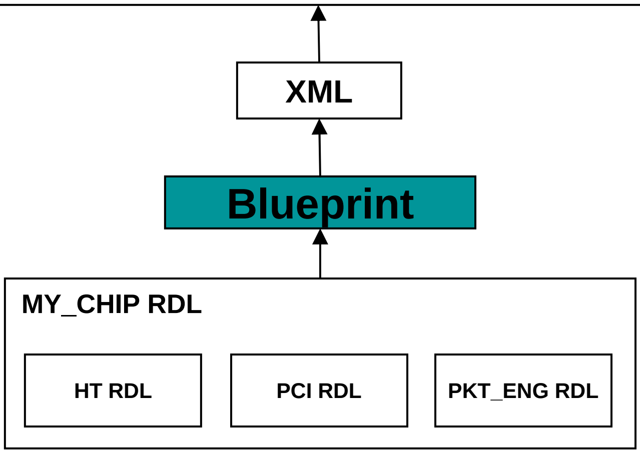
can be any regular expression

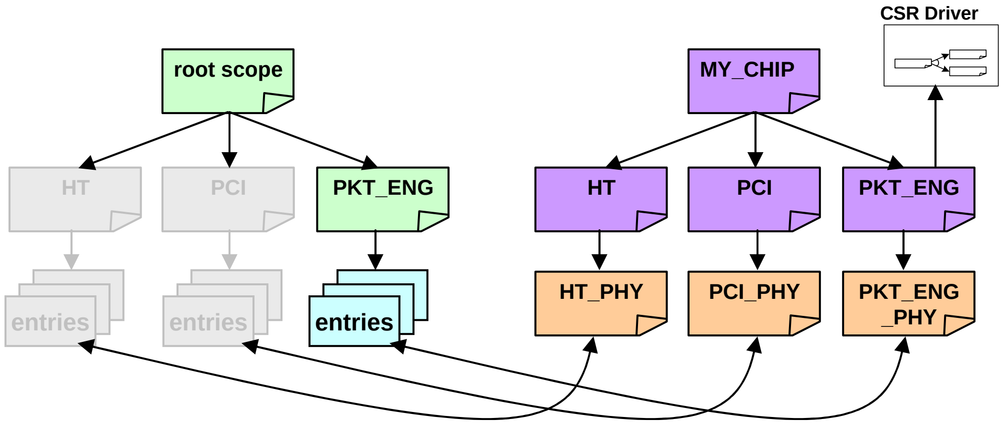
```
extend MY_CHIP csco_addrmap_env_u {
  !bp_region : csco_addrmap_logical_region_s;

  post_generate() is also {
    bp_region = scope.populate_from_xml_file("MY_CHIP.xml", NULL);
  };

};
```

<proj>_common_top.e file – same logic for all testbenches.





Usage Model – Integrating Blueprint

Chip-Level

```
extend MY_CHIP csco_tb_u {
  connect_pointers() is also {
    var bp_region := addrmap.as_a(MY_CHIP csco_addrmap_env_u).blueprint_logical_region;
    ht_env.set_addrmap_region(bp_region);
  };
};
```

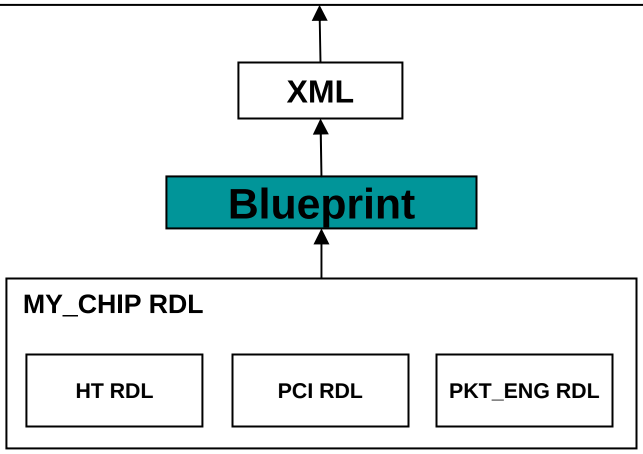
```
extend MY_CHIP csco_tb_u {
  connect_pointers() is also {
    var bp_region := addrmap.as_a(MY_CHIP csco_addrmap_env_u).blueprint_logical_region;
    pci_env.set_addrmap_region(bp_region);
  };
};
```

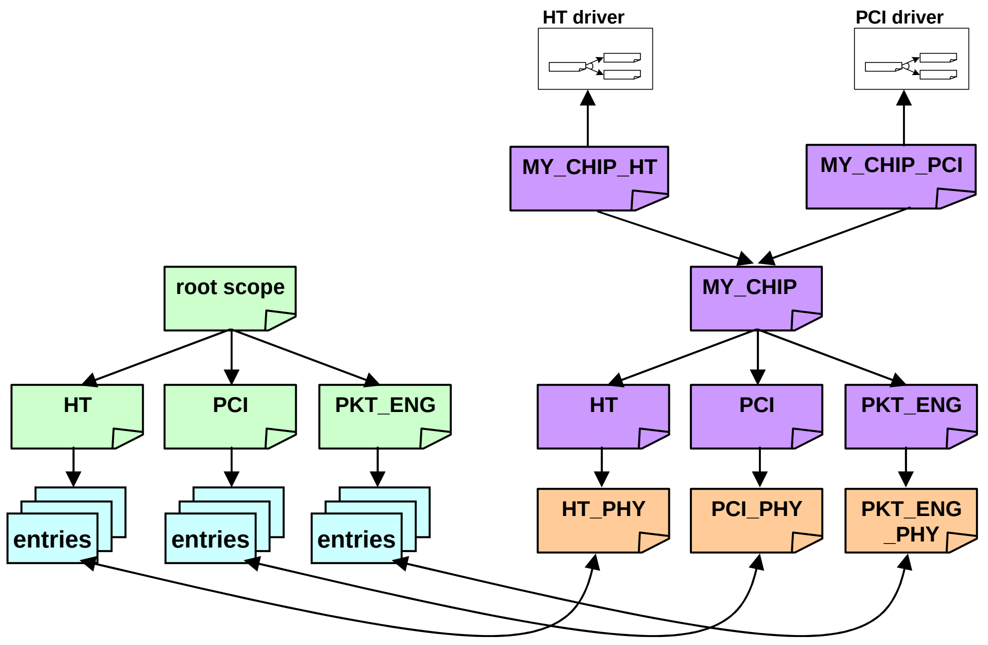
Add access to registers via PCI interface.

```
extend MY_CHIP csco_addrmap_env_u {
  !bp_region : csco_addrmap_logical_region_s;

  post_generate() is also {
    bp_region = scope.populate_from_xml_file(“MY_CHIP.xml", NULL);
  };

};
```





Usage Model – Sequence Interface

```
extend csco_addrmap_seq_kind_t : [PKT_ENG_CFG];

extend PKT_ENG_CFG csco_addrmap_seq {
  pcfg : pkt_eng_config_s;

  body() @driver.clock is also {
    addrmap_do WRITE entry_trans keeping {
      .resp_kind == NON_POSTED;
      .return == BLOCKING;
      .entry == “RING_SIZE”;
      .fields == { .set_field(“Ingress”, pcfg.ig_ring_size);
                   .set_field(“Egress”, pcfg.eg_ring_size) };
    };

    addrmap_do WRITE entry_trans keeping {
      .entry == “RING_CTRL”;
      .fields == { .set_field(“Enable”, 1) };
    };
  };
};
```

Error will result if normal ‘do’ is used

‘entry_trans’ is a predefined field

Usage Model – Interface Driver

called by addrmap driver to forward transactions to interface driver.

```
extend pci_driver_u {
  send_addrmap_item(addrmap_trans: csco_addrmap_trans_s) is {
    csco_addrmap_seq.addrmap_trans =
            addrmap_trans.as_a(CSCO_SPABUS'interface csco_addrmap_trans_s);
    emit csco_addrmap_seq.addrmap_trans_ready;
  };

  addrmap_ready() : bool is {
    result = (csco_addrmap_seq.addrmap_trans == NULL);;
  };
};
```

translation sequence

used to backpressure addrmap sequences targeting this driver.

Only accept 1 addrmap transaction at a time.

Usage Model – Interface Driver

```
extend csco_pci_trans_seq_kind_t : [ CSCO_ADDRMAP ];
extend CSCO_ADDRMAP csco_pci_trans_seq {
  !addrmap_trans : CSCO_PCI'interface csco_addrmap_trans_s;
  event addrmap_trans_ready;
  package !addrmap_segment : csco_addrmap_trans_s;

  body() @driver.clock is {
    while(TRUE) {
      if(addrmap_trans == NULL) {
        sync @addrmap_trans_ready;
      };

      addrmap_segment = addrmap_trans.intf_iterator.get_next_segment();
      addrmap_trans.tracking.incr_exp_child_cnt(1);
      while addrmap_segment != NULL {
        do pci_trans keeping {
          <constrain addrmap segment fields to local transaction fields>;
        };
        <process response if read>;

        addrmap_segment = addrmap_trans.intf_iterator.get_next_segment();
        if addrmap_segment != NULL {
          addrmap_trans.tracking.incr_exp_child_cnt(1);
        };
        addrmap_trans.tracking.link_trans(one_trans);
      };
      addrmap_trans = NULL;
    };
  };
};
```

wait for new addrmap transaction

loop through all segments returned by addrmap iterator

each segment is just another addrmap transaction

signal that we're ready for another addrmap transaction

Usage Model – Configuring Iterator

Cisco.com

```
extend CSCO_SPABUS'interface csco_addrmap_logical_region_s {
  keep legal_alignments.size() == 3;
  keep for each (la) in legal_alignments {
    index == 0 => all of {
      la.alignment == 4;
      la.max_size == 4;
      la.mask_precision == 4;
    };

    index == 1 => all of {
      la.alignment == 2;
      la.max_size == 2;
      la.mask_precision == 2;
    };

    index == 2 => all of {
      la.alignment == 1;
      la.max_size == 1;
      la.mask_precision == 1;
    };
  };
};
```

There can be multiple legal alignments per interface. The addrmap iterator by default uses a greedy algorithm.

User may define custom algorithms for picking amongst segment options.

Usage Model – Shadow Update/Checking

- **Each address map entry maintain shadow values for all its fields.**
- **Each region has a method for updating shadow value in any of its child entries:**

  update_shadow(trans : csco_addrmap_trans_s)

- **Method will:**
  - **check read response against shadow value.**
  - **update shadow value, taking into account side-effects.**

```
extend pci_monitor_u {
  collect_trans_hook(tr : pci_trans_s) is also {
    addrmap_trans = new csco_addrmap_trans_s with {
      .opcode = (tr.dir == READ) ? READ : WRITE;
      .target_region = tb_ptr.addrmap.get_physical_region("PCI");
      .target_offset = tr.address;
      .num_bytes = tr.data.size();
      .data = tr.data;
      .write_mask = tr.mask;
    };
    addrmap_trans.target_region.update_shadow(addrmap_trans);
  };
};
```

Legacy Support

- Task list for migrating legacy processor interface eVCs:
  - sequence driver should extend the ‘send_addrmap_item()’ and ‘addrmap_read()’ methods.
  - create translation sequence that will use each addrmap transaction’s iterator to launch multiple local transactions.
  - extend monitor to call update_shadow() of the appropriate addrmap region whenever a transaction is collected.

CISCO SYSTEMS

EMPOWERING THE INTERNET GENERATION