BSV Training

Eg06: Memory-to-Memory Mergesort

An IP block* that sorts a vector in memory, using the “mergesort” algorithm. 06a uses an *ad hoc* test driver; 06b generalizes this to an “SoC” structure with an AXI4 interconnect; 06c replaces the test driver with a RISC-V CPU core.

*Note: the term “IP block” or “Intellectual Property Block” originates from its reference to non-trivial hardware designs proprietary to a company and which are bought and sold as units. Nowadays the term is also used to refer to any non-trivial hardware block.

www.bluespec.com

Please clone the following REPO, from which we're going to draw some components:

```
$ git clone https://github.com/bluespec/Piccolo
```

In the Makefiles in Eg06a,b,c you will need to edit this definition to point at your clone:

```
# Directory of your clone of https://github.com/bluespec/Piccolo

PICCOLO_REPO   ?= $(HOME)/GitHub/Piccolo
```

Mergesort algorithm

Binary mergesort is a standard sorting algorithm, described in many textbooks and courses on algorithms. The basic idea is illustrated below.

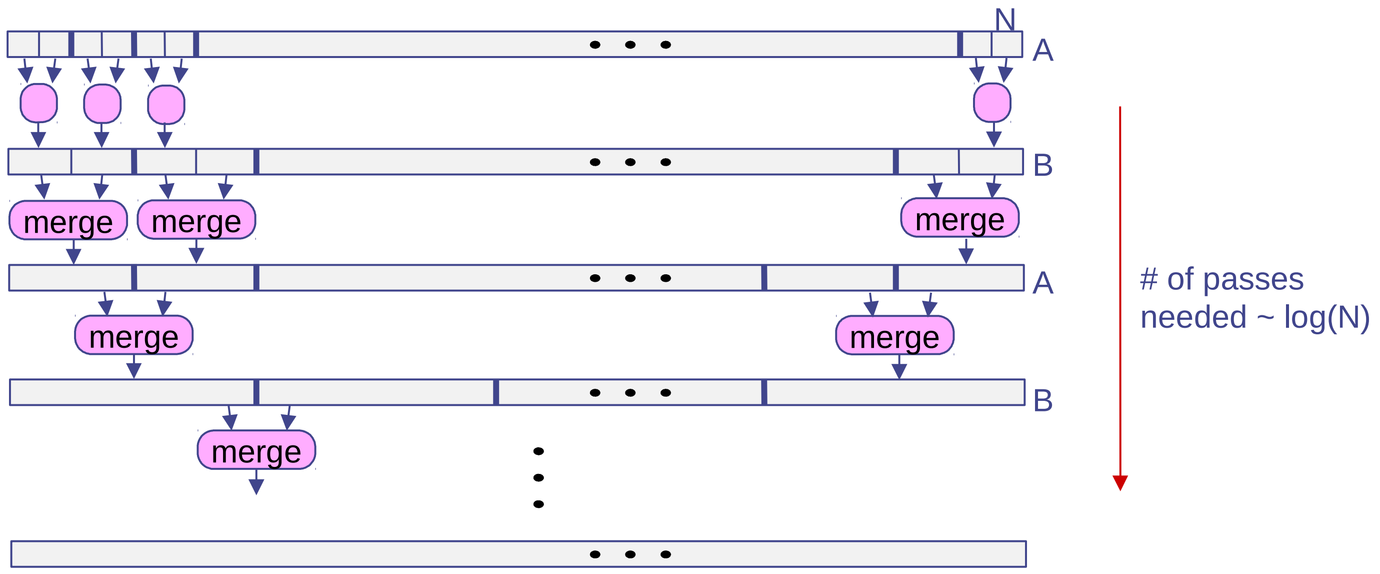
1st pass: merge segments of span 1 into segments of span 2

2nd pass: merge segments of span 2 into segments of span 4

3rd pass: merge segments of span 4 into segments of span 8

•
•
•

… and so on, until segment span >= N, the length of the array



Invariant: every segment that is input to "merge" is already sorted

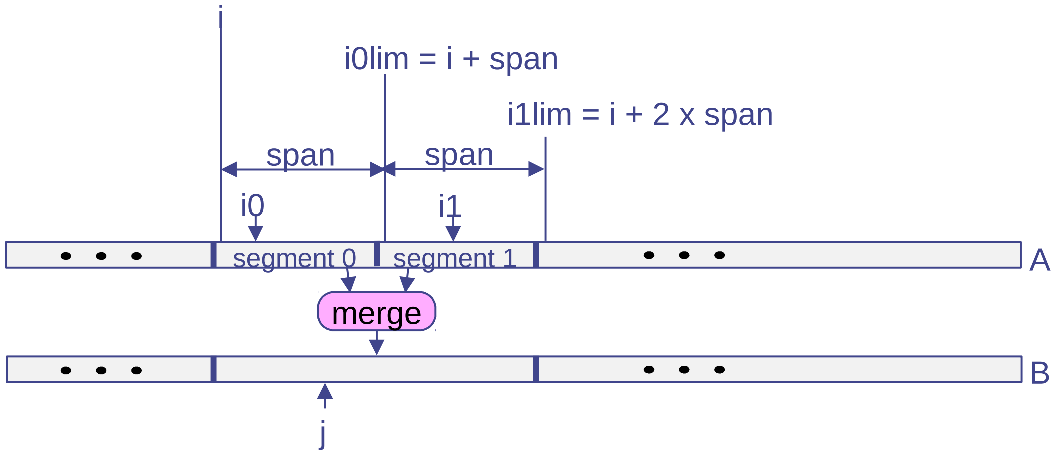
After completing a pass, we can swap arrays A and B.

Some edge conditions we need to take care of:

- N is usually not a power of 2, so last two spans may have unequal length
- Depending on N, the final sorted array may be in B, and so may have to be copied back into the original array A

Mergesort algorithm (contd.)

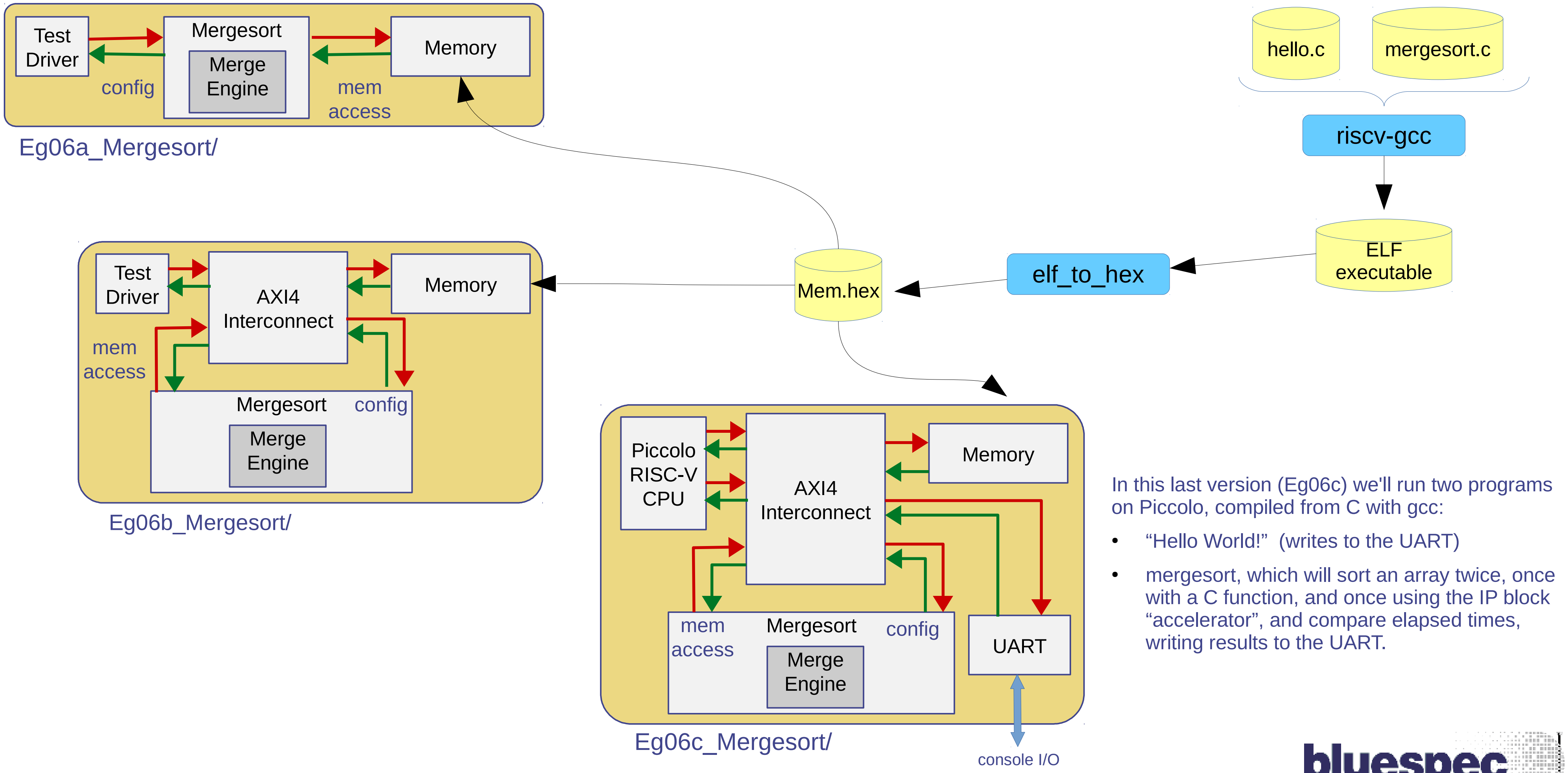
The “merge” step sorts two already-sorted segments of length ‘span’ into a sorted segment of length ‘2 x span’



```
merge_engine (A,B,i,span) {
    i0lim = i + span; i1lim = i + 2*span;
    j = i0 = i; i1 = i0lim;

    while (j < i1lim) {
        if (i0 >= i0lim)         y = A[i1++];   // segment 0 exhausted; take from segment 1
        else if (i1 >= i1lim)    y = A[i0++];   // segment 1 exhausted; take from segment 0
        else if (A[i0] <= A[i1]) y = A[i0++];   // take from segment 0
        else                     y = A[i1++];   // take from segment 1
        B[j++] = y;
    }
}
```

C code

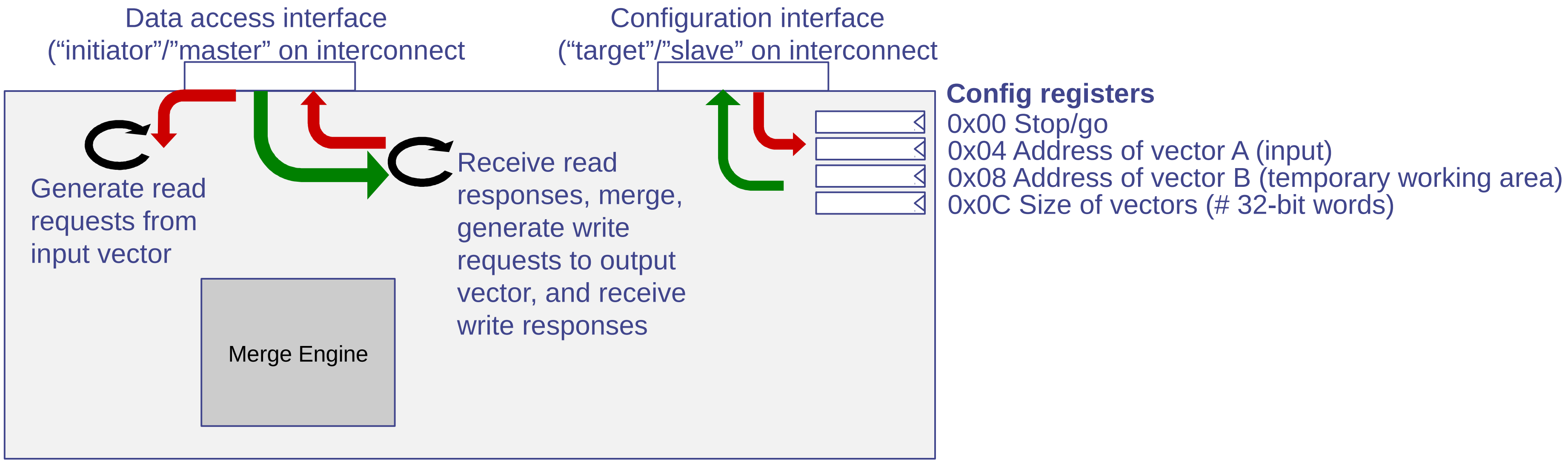
Three variations in this example



In this last version (Eg06c) we'll run two programs on Piccolo, compiled from C with gcc:

- “Hello World!” (writes to the UART)
- mergesort, which will sort an array twice, once with a C function, and once using the IP block “accelerator”, and compare elapsed times, writing results to the UART.

1st version: directory Eg06a_Mergesort/ (contd.)



Data access interface
("initiator"/"master" on interconnect
Configuration interface
("target"/"slave" on interconnect
Generate read requests from input vector
Receive read responses, merge, generate write requests to output vector, and receive write responses
Merge Engine
Config registers
0x00 Stop/go
0x04 Address of vector A (input)
0x08 Address of vector B (temporary working area)
0x0C Size of vectors (# 32-bit words)

To perform the mergesort:

- The external environment must write the addresses and size of the vectors A and B to the config registers at offset 0x04, 0x08 and 0x0C, and finally write a "1" (meaning: "start running") to the config register at offset 0
- The mergesort module then does its work, reading and writing through its data access port; when completed, it writes "0" to the config reg at offset 0
- The external environment can "poll" (repeated read) the config reg at offset 0, to detect completion

Time-out to reinforce some concepts

Please study the lectures:

- Lec_Types to review types, which are used to define memory requests and responses

- Lec_Interfaces_TLM to review the concepts behind interfaces like Get, Put, Client and Server, which are used for most of the interfaces in this example.

- Lec_Interfaces_TLM and Lec_Typeclasses to review the concepts behind the mkConnection abstraction, which is used in the testbench to connect all components together.

- Lec_StmtFSM for the concepts behind structured rule-based processes, which are used both in the mergesort module and in the testbench.

- Lec_Interop_C for the concepts behind importing C code into BSV, which is used in the memory model in this example.

Memory requests: Resources/Req_Rsp.bs

```
-- Operation requested
data RR_Op = RR_Op_R | RR_Op_W
    deriving (Eq, Bits, FShow)

-- Size requested
data RR_Size = RR_Size_8b | RR_Size_16b | RR_Size_32b | RR_Size_64b
    deriving (Eq, Bits, FShow)

-- ----------------
-- Requests
-- Note: wdata is always in the least-significant bits (unlike AXI4!)

struct (RR_Req :: # -> # -> # -> *)  wd_tid  wd_addr  wd_data = {
    tid   :: Bit  wd_tid;    -- Transaction Id
    op    :: RR_Op;
    addr  :: Bit  wd_addr;
    size  :: RR_Size;
    wdata :: Bit  wd_data    -- write-data (not relevant for read-requests)
    }
    deriving (Bits, FShow)
```

Memory requests contain a op (READ/WRITE), an address, data (for WRITE commands), a spec of the size of data being transferred, and a “transaction id” (tid).

Transaction Ids (tids) are common on memory requests/responses in modern SoCs, because:

- There may be multiple initiators, and the tid can serve as a “return address” identifying where a response should go
- Responses may be in a different order from the original requests, and the tid can identify the original order

Note: parameterized on the bit-width of tid, addr, data

Memory requests and responses: Resources/Req_Rsp.bs

```
-- Response status
data RR_Status = RR_Status_OKAY          -- = AXI4 OKAY
               | RR_Status_RESERVED      -- = AXI4 EXOKAY; here unused
               | RR_Status_TARGETERR     -- = AXI4 SLVERR (e.g., misaligned)
               | RR_Status_DECERR        -- = AXI4 DECERR (decode err: no such addr)
    deriving (Eq, Bits, FShow)

-- ----------------
-- Responses
-- Note: rdata is always in the least-significant bits (unlike AXI4!)

struct (RR_Rsp :: # -> # -> *)  wd_tid  wd_data = {
    tid    :: Bit  wd_tid;    -- Transaction Id
    status :: RR_Status;
    rdata  :: Bit  wd_data;   -- read-data (not relevant for write-responses)
    op     :: RR_Op           -- For debugging only
    }
    deriving (Bits, FShow)
```

Memory responses contain the original op (READ/ WRITE), rdata (for READ op), a status, and the original “transaction id” (tid).

Note: parameterized on the bit-width of tid, data

bluespec

1st version: directory Eg06a_Mergesort/ (contd.)

Specific choices for memory requests and responses in our Mergesort example

Resources/Req_Rsp.bs contains generic definitions for the types of memory requests and responses.

In particular, they are parameterized by the bit-widths of addresses (addr_sz), data (data_sz) and transaction ids (tid_sz), so that they can be used in various SoCs with various requirements.

In Eg06a_Mergesort/src/Fabric_Defs.bs, we make particular choices for these parameter for our mergesort example.

```
type Wd_Id = 4

type Wd_Addr = 32

type Wd_Data = 32
```

Tids are 4-bits wide

Addresses are 32-bits wide

Data are 32-bits wide

```
-- Names for types of certain request/response fields

type Fabric_Id   = Bit  Wd_Id
type Fabric_Addr = Bit  Wd_Addr
type Fabric_Data = Bit  Wd_Data
type Fabric_User = Bit  Wd_User

-- Fabric requests, responses
-- (specializations of the generic RR_Req and RR_Rsp)

type Fabric_Req = RR_Req  Wd_Id  Wd_Addr  Wd_Data
type Fabric_Rsp = RR_Rsp  Wd_Id           Wd_Data
```

Specializations of various types for chosen sizes

(in Utils/Fabric_Req_Rsp.bs)

1st version: directory   Eg06a_Mergesort/   (contd.)

Specific choices for memory-mapping in our Mergesort example

In Eg06a_Mergesort/src/SoC_Map.bs, we also make particular choices for the “addresses” at which the memory lives, and at which the mergesort configuration port lives.

```
mem0_controller_addr_base :: Fabric_Addr = 0x80000000
mem0_controller_addr_size :: Fabric_Addr = 0x10000000    -- 256 MB
mem0_controller_addr_lim  :: Fabric_Addr = (  mem0_controller_addr_base
                                            + mem0_controller_addr_size)

accel_0_addr_base :: Fabric_Addr = 0xC0000100
accel_0_addr_size :: Fabric_Addr = 0x00000080    -- 128
accel_0_addr_lim  :: Fabric_Addr = (  accel_0_addr_base
                                    + accel_0_addr_size)
```

1st version: directory Eg06a_Mergesort/ (contd.)

In Mergesort.bs: mkMergeSort module structure

```
mkMergesort :: Module Mergesort_IFC
mkMergesort =
  module
    …
    -- Section: Configuration
    …
    -- Vector of CSRs (Config and Status Regs)
    v_csr :: Vector  N_CSRs  (Reg  Fabric_Addr) <- replicateM  (mkReg 0)
    …
    rules
        "rl_handle_configReq": when True
         ==> do
    …
    -- Section: Merge sort behavior

    merge_engine :: Merge_Engine_IFC <- mkMerge_Engine

    -- 'span' starts at 1, and doubles on each merge pass
    rg_span :: Reg  Fabric_Addr <- mkRegU
    …
    … FSM (in rules) to repeatedly invoke merge_engine with spans 1,2,4,8,…

    interface
        …
        mem_bus_ifc    = merge_engine.mem_bus_ifc
```

Instantiate the configuration registers

This rule receives incoming config requests, reads/writes config regs, sends responses

Instantiate module for the “merge” step

This FSM implements the following pseudo-code:

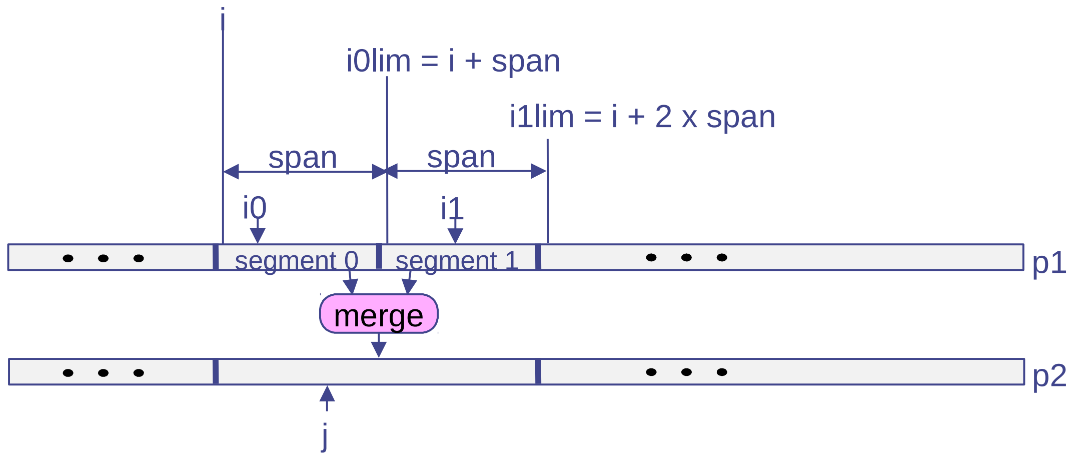
```
while (True)
   wait for ‘run’ command, init span=1, p1=A, p2=B
   while (span < n) // do another pass:
      i=0;
      while (i < n)
         merge_engine (i, span, p1, p2);
         i += 2*span;
      swap p1,p2; span = 2x span
   if final array is B, copy it back to A
   config reg [0] = 0  (announce completion)
```

The mergeEngine’s memory interface is directly used as the memory interface

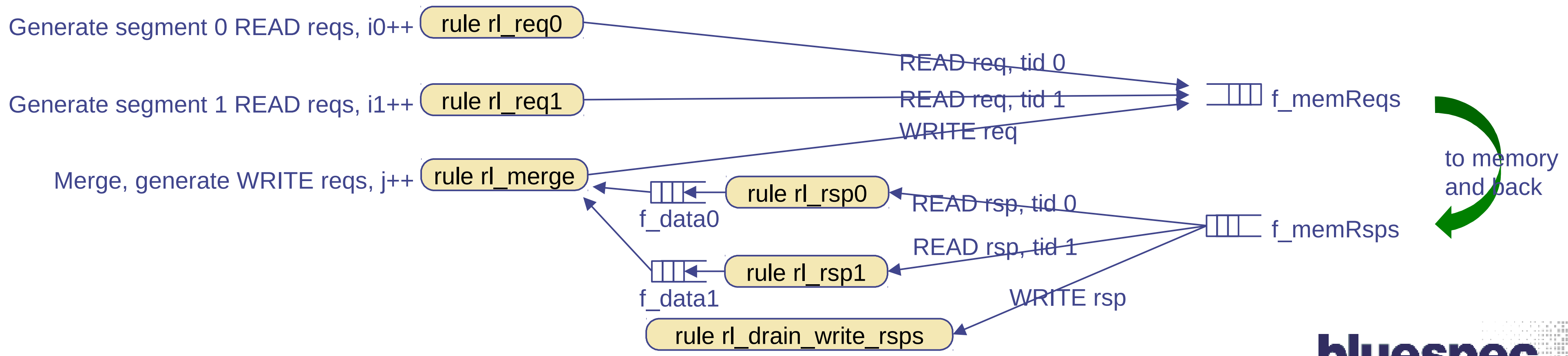
1st version: directory Eg06a_Mergesort/ (contd.)

In Mergesort.bs: mkMergeEngine

This module implements the "merge" step which we saw eariler:



mkMergeEngine module data flow



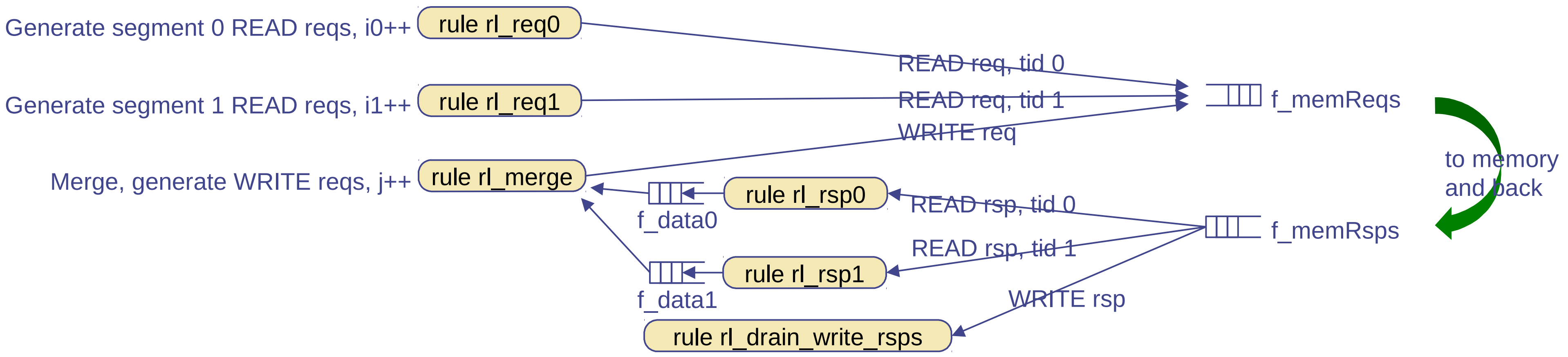
1st version: directory Eg06a_Mergesort/ (contd.)

mkMergeEngine is highly concurrent (pipelined):

- rl_req0, rl_req1 and rl_merge continuously stream requests to memory
- rl_rsp0, rl_rsp1 and rl_drain… continuously handle the stream of responses

This is typical of high-performance accelerators which try to maximize utilization of available memory bandwidth. A software implementation on a CPU may not be able to generate such concurrent, pipelined memory accesses.

mkMergeEngine module data flow



1st version: directory Eg06a_Mergesort/ (contd.)

There is a danger of deadlock. *Example:*

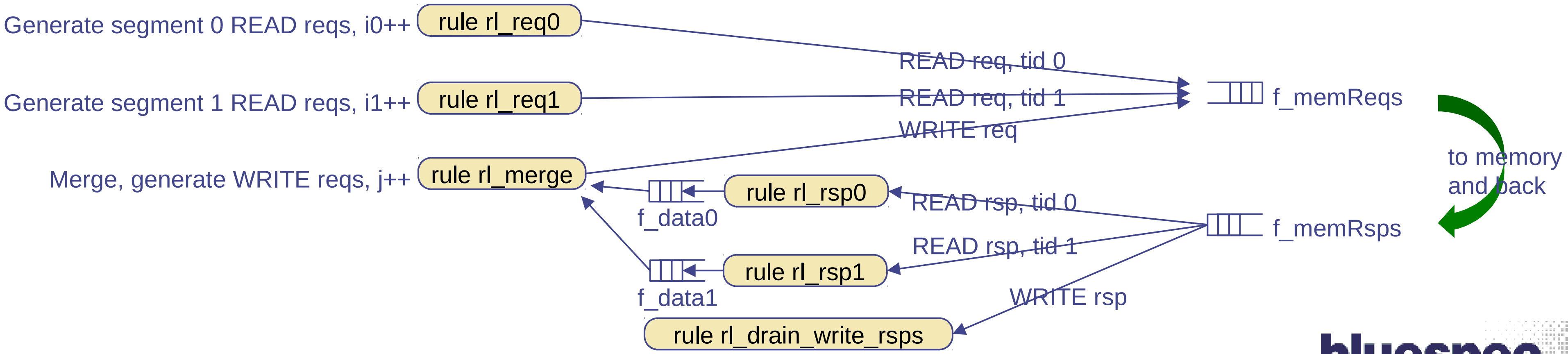
- Suppose rl_merge does not consume f_data1, for example because the next segment 1 item is > many segment 0 items
- Then, f_data1 may become full, and if the first item in f_memRsps is from segment 1, then we get stuck (the segment 0 items we need may be behind it). This kind of deadlock is called “head-of-line blocking”

Solution:

- The code has a parameter: max_n_reqs_in_flight = 8
- f_data0 and f_data1 are sized to accommodate 8 responses. “Credit counters” crg_credits0 and crg_credits1 are also intialized to 8.
- rl_req0 and rl_req1 decrement crg_credits0 and crg_credits1, respectively, whenever they issue a memory request, and stop issuing requests when their credit goes to 0.
- rl_merge increments crg0_credits0 (respectively, crg1_credits) whenever it consumes a response from f_data0 (respectively, f_data1)
- Using a CReg (instead of a Reg) allows a {rl_req0, rl_req1} and rl_merge to operate concurrently (increment and decrement credits in the same cycle).

This prevents the above deadlock situation.

mkMergeEngine module data flow



1st version: directory Eg06a_Mergesort/ (contd.)

In Resources/Memory_Model.bs: a memory model

To test our mergesort block, we need to provide a memory containing the vector A to be sorted and the vector B for its scratch working area.

Large memories (particularly those implemented in DRAM) are typically not expressed in a hardware design language. Hence we merely use a *model* of memory for testing our IP block in simulation.

This is provided in Resources/Memory_Model.bs, which is excerpted below:

```
interface Memory_IFC =
   bus_ifc    :: Server  Fabric_Req  Fabric_Rsp
   …

mkMemory_Model :: Module Memory_IFC
mkMemory_Model =
  module
    …
```

The interface is a "Server" where one can "put" Fabric_Reqs and "get" Fabric_Rsps.

The body of the module mkMemory_Model is fairly straightforward. We use a Bluespec "Register File" to model memory, in particular a variant that will load its initial contents from a "Mem.hex" file:

```
    let last_index :: Raw_Mem_Addr = 0x4000000 - 1    -- 16M Raw_Mem_Words

    rf :: RegFile  Raw_Mem_Addr  Raw_Mem_Word <- mkRegFileLoad  "Mem.hex"
                                                                 0
                                                                 last_index
```

The register file is 4-bytes wide (to accommodate the common access pattern), but it also supports 1- and 2-byte accesses.

It also checks that access requests are naturally aligned.

1st version: directory Eg06a_Mergesort/ (contd.)

In Testbench.bs: a testbench

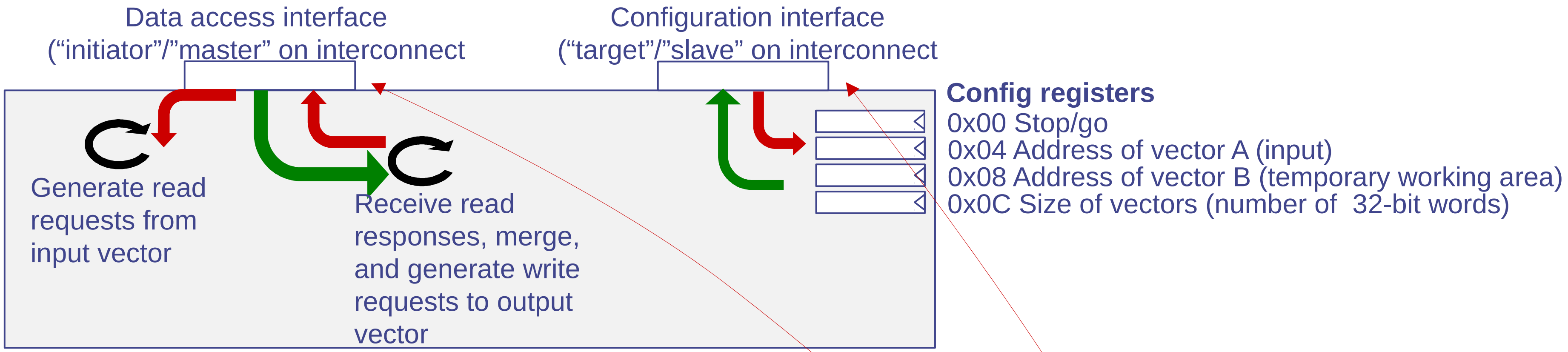
Our testbench is excerpted below:

```
interface Test_Driver_IFC =
    start   :: Action
    busy    :: Bool
    bus_ifc :: Client  Fabric_Req  Fabric_Rsp

mkTest_Driver :: Module  Test_Driver_IFC
mkTest_Driver =
  module
   …
    rules
         … write mergesort’s config regs …
         … loop, polling mergesort’s config reg for completion …
```

The testbench only performs a very small test (sort 29 words) so that you can easily inspect the outputs:

1st version: directory Eg06a_Mergesort/ (contd.)

Interface for our mergesort module


Data access interface
("initiator"/"master" on interconnect
Configuration interface
("target"/"slave" on interconnect
Generate read requests from input vector
Receive read responses, merge, and generate write requests to output vector

Config registers

0x00	Stop/go
0x04	Address of vector A (input)
0x08	Address of vector B (temporary working area)
0x0C	Size of vectors (number of 32-bit words)

In file Mergesort.bs:

```
-- Interface of Mergesort module
-- The init arg 'addr_base' is the base address in an SoC for its config regs

interface Mergesort_IFC =
    init          :: Fabric_Addr -> Action
    config_bus_ifc :: Server  Fabric_Req  Fabric_Rsp
    mem_bus_ifc    :: Client  Fabric_Req  Fabric_Rsp
```

bluespec

Build and run the 1st version

Please clone the following REPO, from which we're going to draw some components:

```
$ git clone https://github.com/bluespec/Piccolo
```

In the Makefiles in Eg06a,b,c you will need to edit this definition to point at your clone:

```
# Directory of your clone of https://github.com/bluespec/Piccolo

PICCOLO_REPO   ?= $(HOME)/GitHub/Piccolo
```

Do this, to get a needed component:

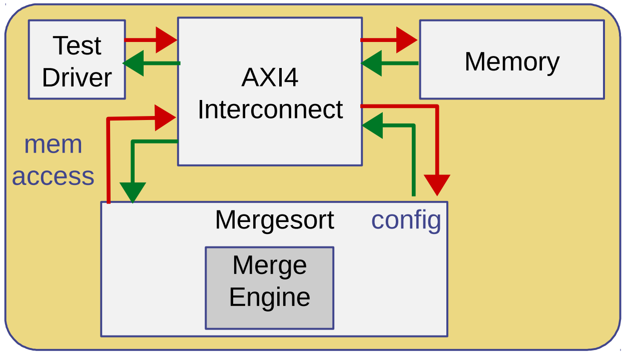
```
$ make  copy_files
```

Build and run using the 'make' commands, with Bluesim and/or with Verilog sim, as described earlier

Observe the inputs and outputs and verify that they are reasonable (final memory contents are a sorted version of initial memory contents)

2st version: directory Eg06b_Mergesort/

Let us prepare our module to be ready for plugging into an “SoC” (System on a Chip) as an “accelerator” module, illustrated to the right.



Eg06b_Mergesort/

An SoC typically consists of CPUs, memories, an interconnect, and custom IP blocks (“Intellectual Property Blocks”) that perform particular functions for reasons of greater speed (acceleration) and/or less power consumption (compared to executing the same function in software on a CPU).

The interconnect fabric carries memory requests (red paths in the figure) and responses (green paths).

- *Initiator* ports (like the CPU port and the IP block read/write port) send requests and receive responses
- *Target* ports (like the Memory port and the IP block config port) receive requests and send responses

Memory requests are routed to the memory or to the IP block config port based on the address contained in the request. I.e., the (usually small number of) configuration registers in the IP block appear, to the CPU, just like memory locations at a particular base address (these addresses are disjoint from addresses serviced by the Memory block). We also say that the config registers are “memory-mapped”.

To operate the IP block, the CPU writes information needed for the operation to the config registers in the IP block, after which the IP block can perform its function (by reading and writing to memory). When the function is completed, the IP block may write a particular value to one of its config registers. The CPU can detect when the IP block has completed its function by “polling” (repeatedly reading) this register.

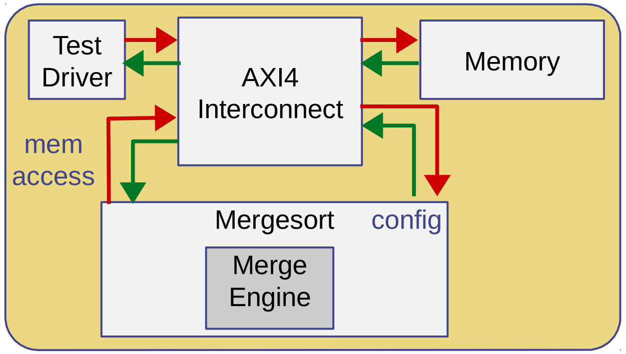
[In practice, IP blocks can also “interrupt” the CPU on completion; our examples here do not do this.]

bluespec

2nd version: directory Eg06b_Mergesort/

In this version we re-use components from Eg06a (Test Driver, Mergesort, Memory).

We only generalize the environment around it into a “SoC” model.



Eg06b_Mergesort/

We will get the AXI4 interconnect from Bluespec's Piccolo RISC-V repository.
Please clone this now:

```
$ git clone https://github.com/bluespec/Piccolo
```

Use 'make copy_files' to copy re-used components from Eg06a/src/ to Eg06b/src/
and to copy AXI4 files from the Piccolo repo:

```
# Please edit the definition of PICCOLO_REPO in Makefile to point at your Piccolo clone directory.

$ make copy_files
```

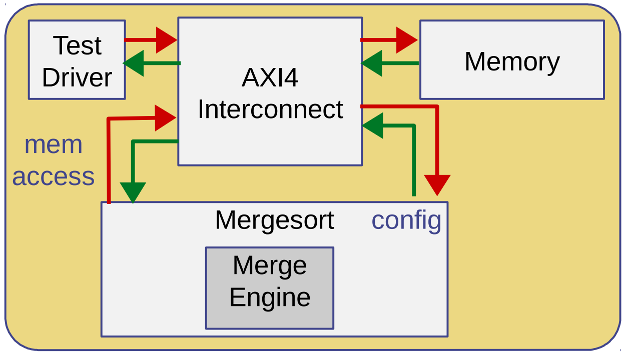
2nd version: directory Eg06b_Mergesort/

Please study: src/SoC_Map.bs

which describes the overall "address map" for the SoC,

and the number of "initiators" and "targets" on the interconnect fabric.

- Initiators (a.k.a. Clients, Masters): send requests, receive responses
  - (Test Driver, Mergesort's memory-access port)
- Targets (a.k.a. Servers, Slaves): receive requests, send responses
  - (Memory, Mergesort's configuration port)

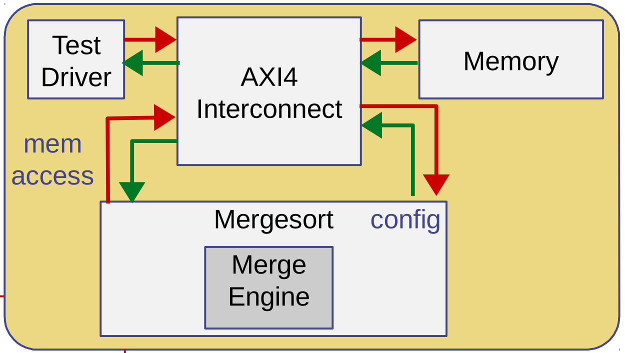


Eg06b_Mergesort/

```
-- Count and initiator-numbers of initiators in the fabric.

type Num_Initiators = 2

test_driver_initiator_num :: Integer; test_driver_initiator_num = 0
accel_0_initiator_num     :: Integer; accel_0_initiator_num     = 1

-- Count and target-numbers of targets in the fabric.

type Num_Targets = 2
type Target_Num  = Bit  (TLog  Num_Targets)

mem0_controller_target_num :: Integer;  mem0_controller_target_num = 0
accel_0_target_num         :: Integer;  accel_0_target_num         = 1
```

Note that we do some "type-level" arithmetic to derive the type of a target number, based on the number of targets.

2nd version: directory Eg06b_Mergesort/



Please study: src/AXI4_Fabric.bsv

It’s interface is just a vector of Servers facing the initiators,
and a vector of Clients facing the targets:

```
interface AXI4_Fabric_IFC #(numeric type num_masters,
                            numeric type num_slaves,
                            numeric type wd_id,
                            numeric type wd_addr,
                            numeric type wd_data,
                            numeric type wd_user);
   method Action reset;
   method Action set_verbosity (Bit #(4) verbosity);

   // From masters
   interface Vector #(num_masters, AXI4_Slave_IFC #(wd_id, wd_addr, wd_data, wd_user))  v_from_masters;

   // To slaves
   interface Vector #(num_slaves,  AXI4_Master_IFC #(wd_id, wd_addr, wd_data, wd_user)) v_to_slaves;
endinterface
```

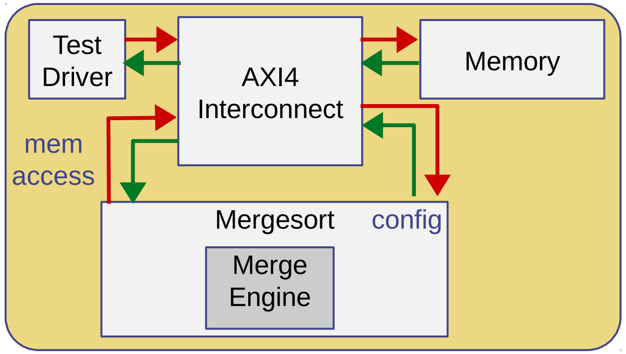
The module mkAXI4_Fabric is a “full crossbar” switch, i.e., there is a separate datapath from each of M initiators to each of S targets, and vice versa. These are represented by rules that are generated in for-loops.
The whole module is parameterized by a “routing function” that decides, based on the address in each request, which target (if any) it should be sent to.

2nd version: directory Eg06b_Mergesort/

Please study: src/SoC_Fabric.bs
to see how we specialize the general AXI4 Fabric definition for our particular SoC.

Inside the module we have an "address decoder" function that decides, based on the address in a request, which target (if any) it should be sent to.
This function is passed as an argument to the mkAXI4_Fabric module constructor.

```
mkSoC_Fabric :: Module  SoC_Fabric_IFC
mkSoC_Fabric =
  module
    soc_map :: SoC_Map_IFC <- mkSoC_Map

    -- Target address decoder.
    -- Identifies whether a given addr is legal and, if so, which target services it.
    let fn_addr_to_target_num :: Fabric_Addr -> (Bool, Target_Num)
        fn_addr_to_target_num    addr =
            -- Mem 0
            if (   (soc_map.m_mem0_controller_addr_base <= addr)
                && (addr < soc_map.m_mem0_controller_addr_lim)) then
                (True, fromInteger mem0_controller_target_num)

            -- Accelerator 0
            else if (   (soc_map.m_accel_0_addr_base <= addr)
                     && (addr < soc_map.m_accel_0_addr_lim)) then
                (True, fromInteger accel_0_target_num)

            else
               (False, _ )

    soc_fabric :: SoC_Fabric_IFC <- mkAXI4_Fabric  fn_addr_to_target_num

    return soc_fabric
```



Please study:

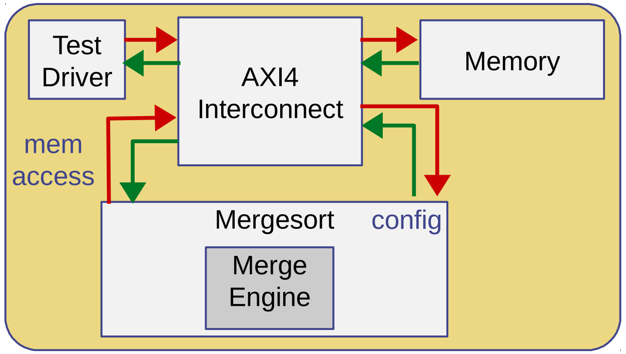
- src/AXI4_Types.bsv
  to see how we define industry-standard AXI4 bus types and interfaces in Bluespec code.
- src/AXI4_Fabric.bsv
  to see how we define an AXI4 crossbar switch that is parameterized by number of initiators, number of targets, and width of the address, data, id and user buses.

These are written in BSV, not Bluespec Classic. AXI4_Types is more easily written in BSV (because of the heavy use of Verilog signal-naming customization). AXI4_Fabric could just as easily be written in Classic.

2nd version: directory   Eg06b_Mergesort/

To re-use a component like the Test Driver,
we use adapters defined in: Resources/Adapters_Req_Rsp_AXI4.bs.

Then, in: src/Top.bs
after instantiating Test_Driver,
we convert its original Req_Rsp interface into and AXI4_Master interface:



```
test_driver :: Test_Driver_IFC <- mkTest_Driver
…
test_driver_master :: SoC_Fabric_Initiator_IFC <- mkReq_Rsp_to_AXI4_Master  test_driver.bus_ifc
```

Then, we connect the AXI4 Master interface to one of the fabric's “sockets”:

```
mkConnection  test_driver_master  (soc_fabric.v_from_masters !! test_driver_initiator_num)
```

Similarly, we use adapters on mergesort's mem_bus interface and config_bus interface, and the memory's bus interface,
and connect them all up to the fabric using 'mkConnection'.

Build and run the 2nd version

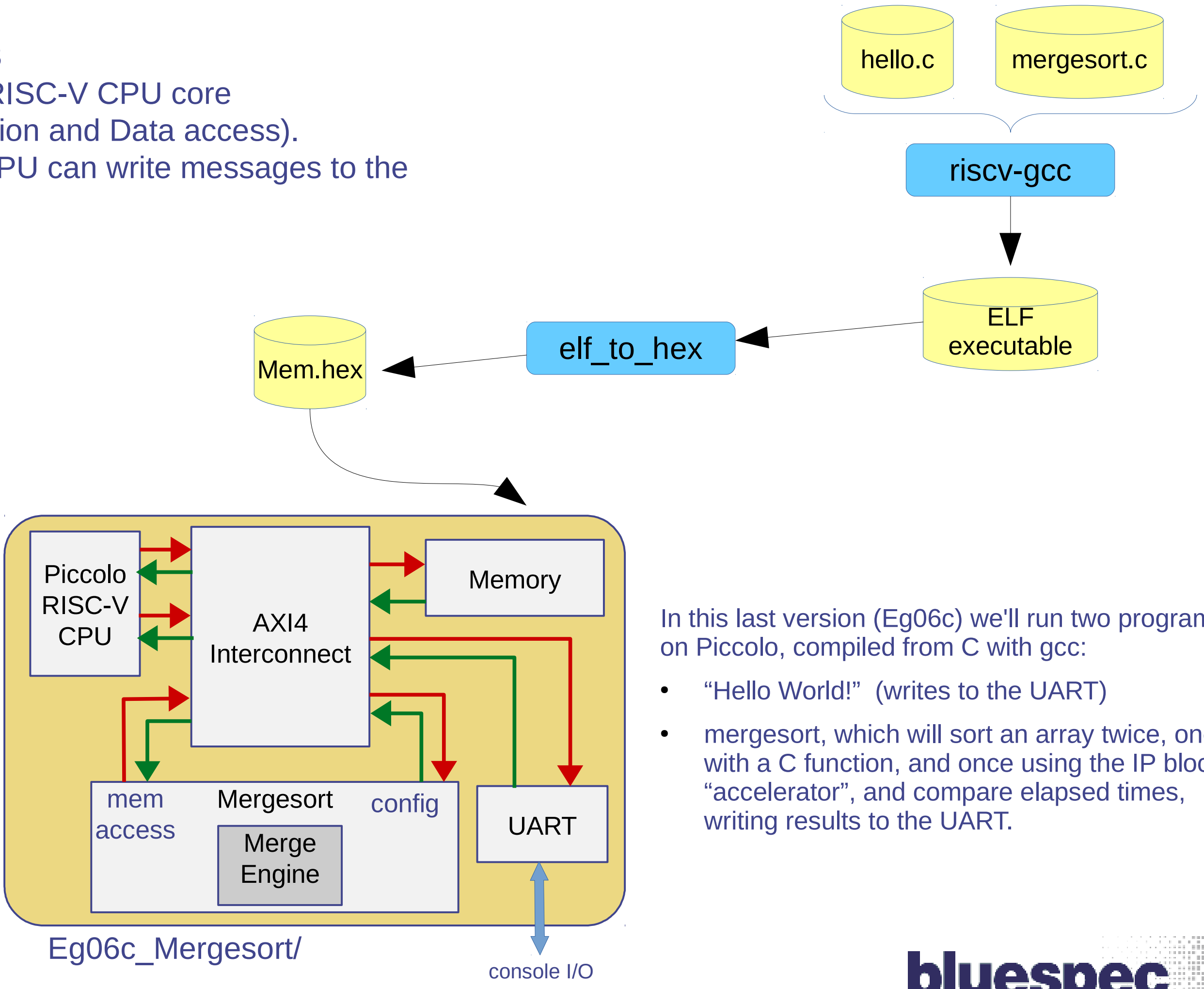
Build and run using the ‘make’ commands, with Bluesim and/or with Verilog sim, as before, and verify that the output is as expected.

3rd version: directory Eg06c_Mergesort/

In this version:

- We expand the AXI4 interconnect from 2x2 to 3x3
- We replace Test_Driver with a Bluespec Piccolo RISC-V CPU core
- It has two initiator ports into the fabric (for Instruction and Data access).
- We add a UART target to the fabric, so that the CPU can write messages to the console.

Then, we run programs on Piccolo by pre-loading the binary program code into memory from a "Mem.hex" file. The binary program code, in turn, is derived from the compiled ELF file for the program.



In this last version (Eg06c) we'll run two programs on Piccolo, compiled from C with gcc:

- "Hello World!" (writes to the UART)
- mergesort, which will sort an array twice, once with a C function, and once using the IP block "accelerator", and compare elapsed times, writing results to the UART.

bluespec

3rd version: directory Eg06c_Mergesort/ (contd.)

Please clone Bluespec's Piccolo RISC-V repository if you had not already done so in Eg06b.

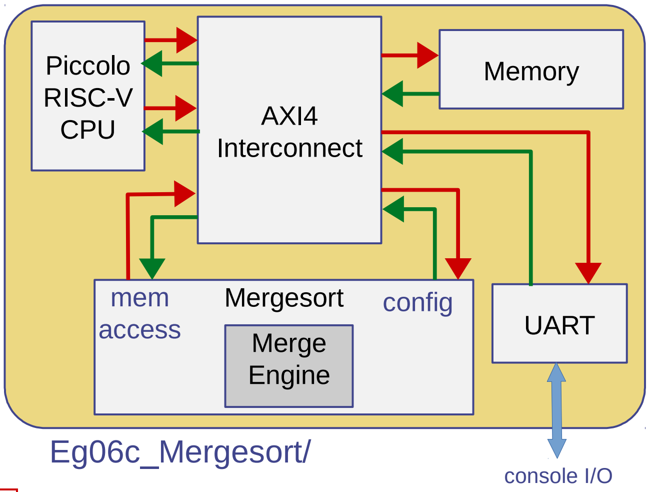
```
$ git clone https://github.com/bluespec/Piccolo
```

Copy components from Eg06a into local copies:

```
$ make copy_files
```

Copy Piccolo files (this will copy into a new "src_Piccolo" directory:

```
# Please edit the definition of PICCOLO_REPO in Makefile to point at your Piccolo clone directory.

$ make copy_Piccolo_files
```



Eg06c_Mergesort/

Build (compile and link) as before:

```
$ make  b_compile  b_link
$ make  b_sim
```

Note: since we haven't provided a Mem.hex file to load into memory, the CPU sees and illegal instruction and falls into an infinite trap loop.

Study the source codes of the two programs of interest: ../C_programs_RV32/hello/hello.c and …/mergesort/mergesort.c

Run the two programs of interest:

```
$ make  b_sim_hello

$ make  b_sim_mergesort
```

First copies ../C_programs_RV32/hello/hello_Mem.hex to Mem.hex; then runs.

First copies ../C_programs_RV32/mergesort/mergesort_Mem.hex to Mem.hex; then runs.

Suggested exercises

- All three versions of the example sort 32-bit (4-byte) words of memory.
  - Modify the design to have a *static* parameter such that it will compile to a circuit that sorts memory in units of 1, 2, 4 or 8 bytes (static = the size is fixed at compile time). Note that Resources/Req_Rsp.bs already defines an enum type RR_Size to specify byte size.
    - The mergesort engine should issue memory requests with the selected unit size.
  - Modify the design so that the byte-size selection is done *dynamically*:
    - Add another config register in which the CPU can specify the size.
    - The mergesort engine should issue memory requests with the selected unit size.
  - Modify the last design so that memory requests are always for 8 bytes, even if the sort is on smaller units. E.g., if the sort is on 1-byte units:
    - Only 1 memory read is needed to fetch 8 units.
    - Only 1 memory write is needed to store 8 units.

- All the examples perform a *binary* (radix 2) merge sort, i.e., the basic merge step merges *two* spans.
  - Modify the program to perform a radix 4 merge sort, i.e., the basic merge step should merge *four* spans. Question: when sorting an array of length *n*, how many memory references does this perform, compared to the binary merge sort?
  - Parameterize the module for a radix *k* mergesort, where *k* is a static parameter that may take some chosen range of values (2, 3, 4, …).

- mkMergesort currently instantiates 1 copy of mkMerge_Engine. If your fabric and memory have more bandwidth, it could instantiate multiple mkMergeEngines running concurrently to do each pass faster. Make this modification: expand the fabric to 4x4; add a second memory bank as the extra target. Instantiate 2 mkMerge_Engines, using the extra initiator port for the second one. Modify mkMergesort to use two engines concurrently. (Caution! having two memory banks can introduce a memory-ordering problem! Use a “CompletionBuffer” from the Bluespec library to solve this.)

Summary

This example has shown you key features of an IP block built for high-performance in an SoC context:

- Useful functionality (sorting, which is useful in *many* applications)
- Implementation using an efficient mathematical algorithm (mergesort)
- Key concepts of SoC structure: Fabrics, initiators, targets, memory mapping, …
- Key concepts of high-performance: pipelining, task queue parallelism, memory bandwidth, managing out-of-order communication, …
- Generality through parameterization on many dimensions (and hence capable of much re-use in other contexts)

End