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

Mergesort algorithm

Binary mergesort is a standard sorting algorithm, described in many textbooks and courses on algorithms. The basic idea is illustrated below.
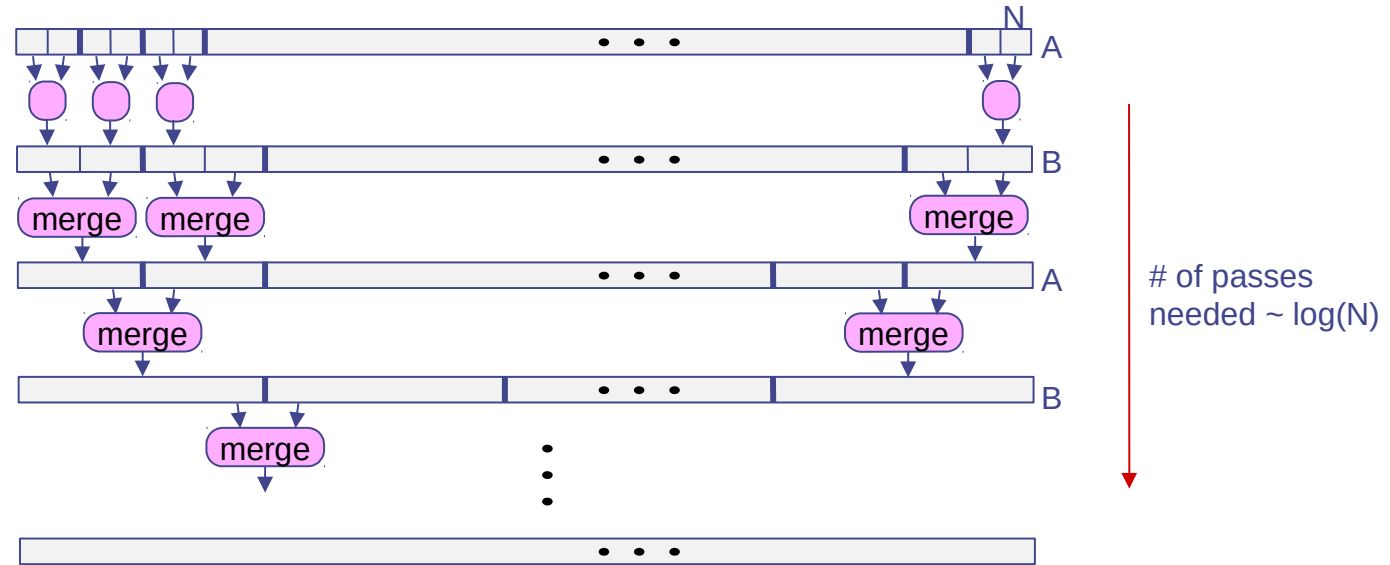
1st pass: merge segments of span 1 into segments of span 2

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

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

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

# of passes needed ~ log(N)

If we complete one pass before starting the next, we can alternate between two arrays A and B.

2

bluespec

Mergesort algorithm (contd.)

The "merge" step sorts two already-sorted segments of length 'span' into a sorted segment of length '2 x span'
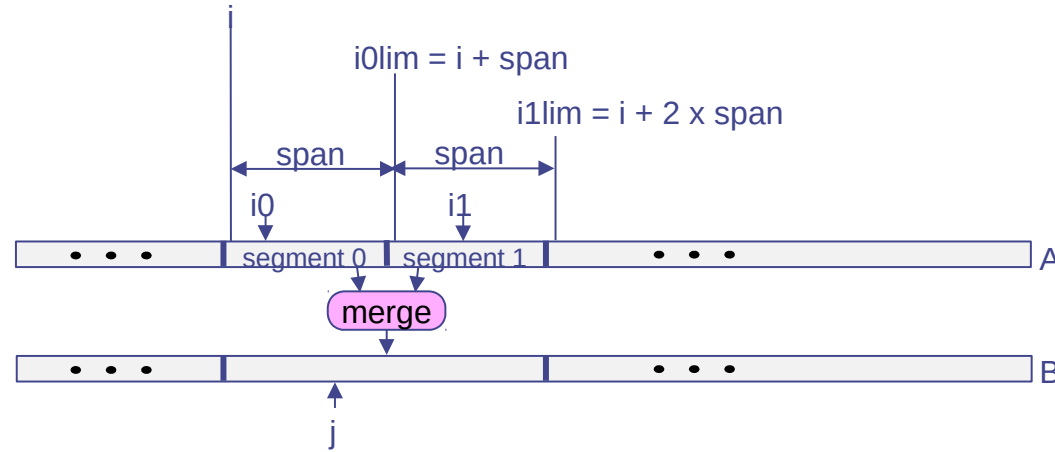

```
merge_engine (A,B,i,span) {
    i0lim = i + span; i1lim = i + 2*span;
    j = i0 = i; i1 = i0lim;

    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;
    }
}
```

bluespec

Example variations

The accompanying code demonstrates three variations:

Eg06a_Mergesort/	Mergesort using a single "merge engine". Testbench connects this to a single-port memory model.
Eg06b_Mergesort/	Generalizes system structure to an "SoC" structure: an AXI4 interconnect fabric with the mergesort IP module as a programmable "accelerator", the test driver as an initiator and the memory model as a target.
Eg06c_Mergesort/	Replaces the test driver with a RISC-V CPU core. Mergesort can now either be run in software on the RISC-V, or using the hardware IP as an accelerator.

bluespec

Rather than create an *ad hoc* interface for our mergesort module, let us prepare it to be ready for plugging into an "SoC" (System on a Chip) as an "accelerator" module, illustrated to the right.

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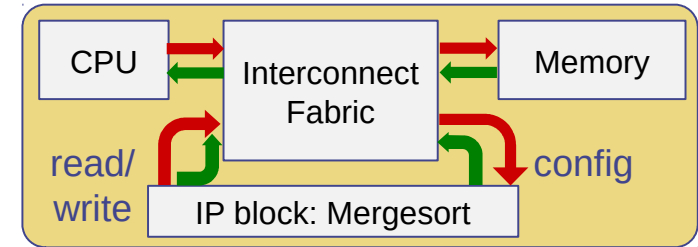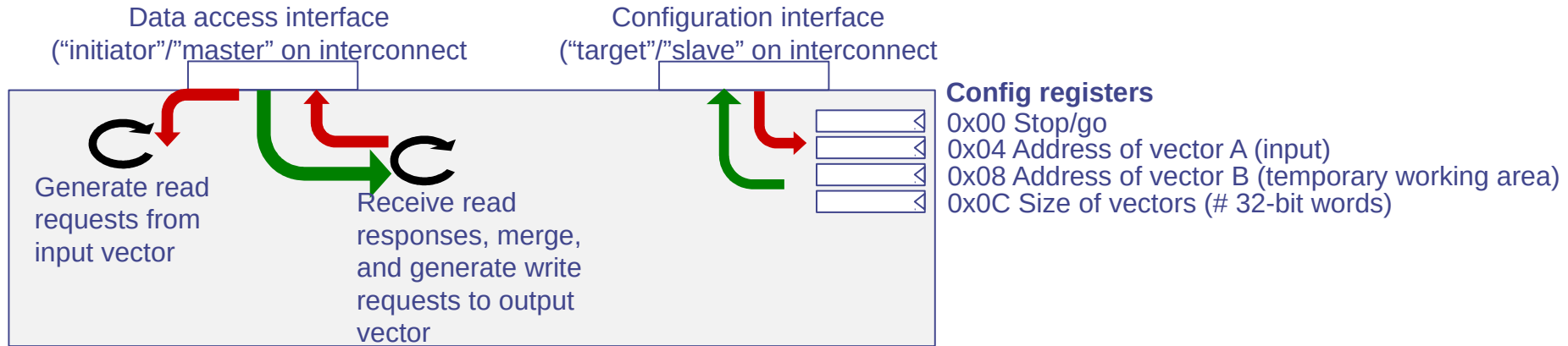
The interconnect fabric carries memory requests (red paths in the figure) and responses (green paths).

- *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

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 (# 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

bluespec

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.

bluespec

```
-- 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

bluespec

```
-- 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

9

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
```

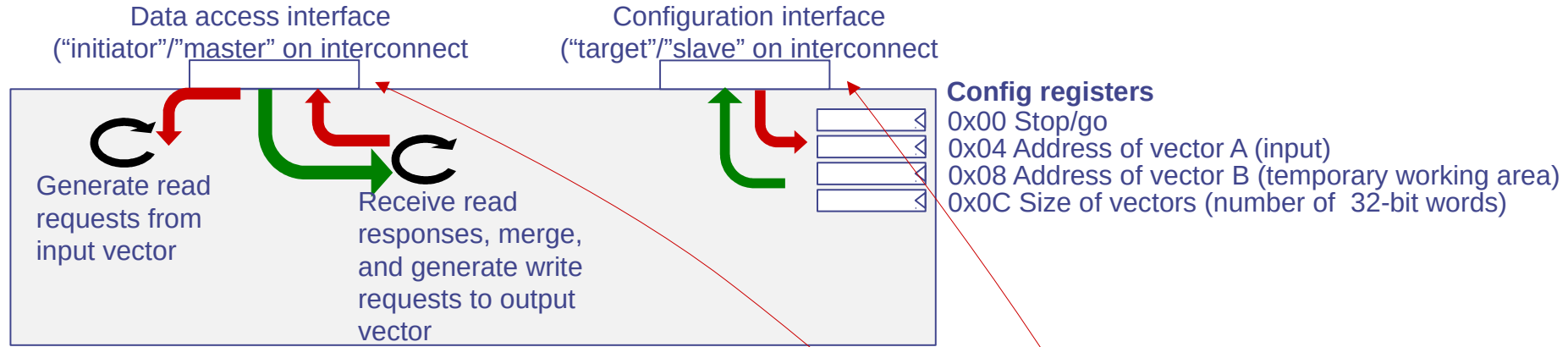Specializations of various types for chosen sizes

(in Utils/Fabric_Req_Rsp.bs)

bluespec

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)
```

bluespec

Interface for our mergesort module

Data access interface
("initiator"/"master" on interconnect

Configuration interface
("target"/"slave" on interconnect

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)

Generate read requests from input vector

Receive read responses, merge, and generate write requests to output vector

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

In Mergesort.bs: mkMergeSort module structure

```
module mkMergeSort (Mergesort_IFC);

    // -------------------------------------
    // Section: Configuration
    …
    Vector #(N_Config_Regs, Reg #(Data)) vrg_configs;
    …
    rule rl_handle_configReq;

        …
    endrule


    // -------------------------------------
    // Section: Merge sort behavior
    …
    MergeEngine_IFC   mergeEngine <- mkMergeEngine;
    …
    mkAutoFSM (
        seq

            …
        endseq);

    // -------------------------------------
    // INTERFACE
    …
    interface mem_bus_ifc = mergeEngine.mem_bus_ifc;
endmodule
```

Instantiate the configuration registers

This rule receives incoming config requests, reads/writes config regs, sends responses
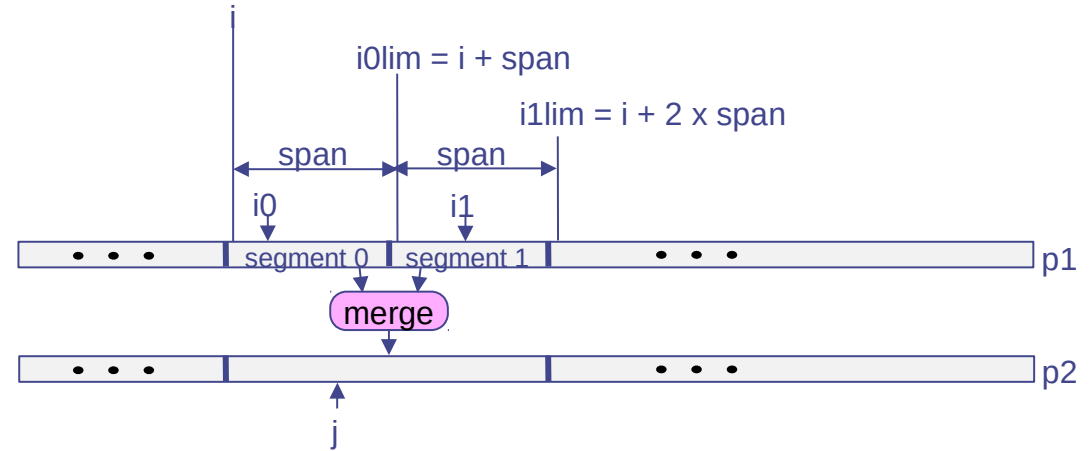
Instantiate module for the "merge" step

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 (i, span, p1, p2);
                i += 2*span;
            swap p1,p2; span = 2x span
        if final array is B, copy it back to A
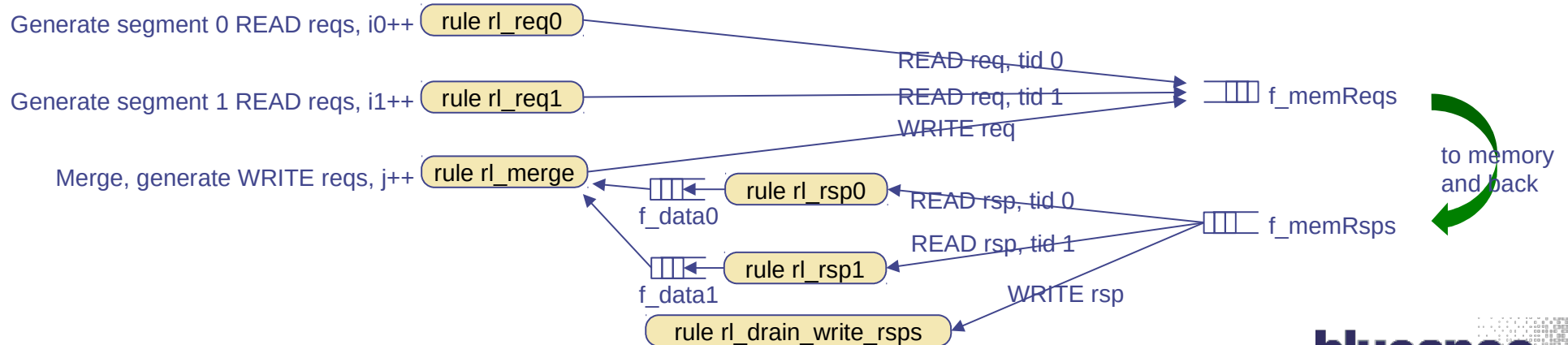   config reg [0] = 0 (announce complete)

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

bluespec

In Mergesort.bs: mkMergeEngine

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

$i0lim = i + span$

$i1lim = i + 2 \times span$

span   span

i0   i1

· · ·   segment 0 | segment 1   · · ·   p1

merge

· · ·   · · ·   p2

j

mkMergeEngine module data flow

Generate segment 0 READ reqs, i0++   rule rl_req0

Generate segment 1 READ reqs, i1++   rule rl_req1

Merge, generate WRITE reqs, j++   rule rl_merge

READ req, tid 0

READ req, tid 1

WRITE req

f_memReqs

rule rl_rsp0   READ rsp, tid 0

f_data0

rule rl_rsp1   READ rsp, tid 1

f_data1

rule rl_drain_write_rsps   WRITE rsp

f_memRsps

to memory and back

bluespec

mkMergeEngine is highly concurrent:

- 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

Generate segment 0 READ reqs, i0++   rule rl_req0

READ req, tid 0

Generate segment 1 READ reqs, i1++   rule rl_req1

READ req, tid 1

WRITE req

f_memReqs

to memory and back

Merge, generate WRITE reqs, j++   rule rl_merge

f_data0   rule rl_rsp0

READ rsp, tid 0

f_memRsps

READ rsp, tid 1

f_data1   rule rl_rsp1

WRITE rsp

rule rl_drain_write_rsps
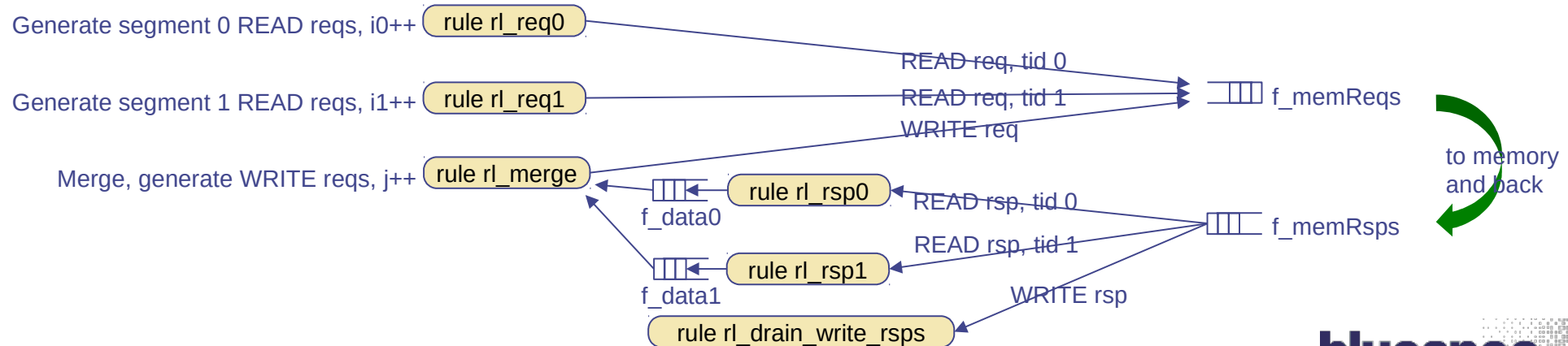
bluespec
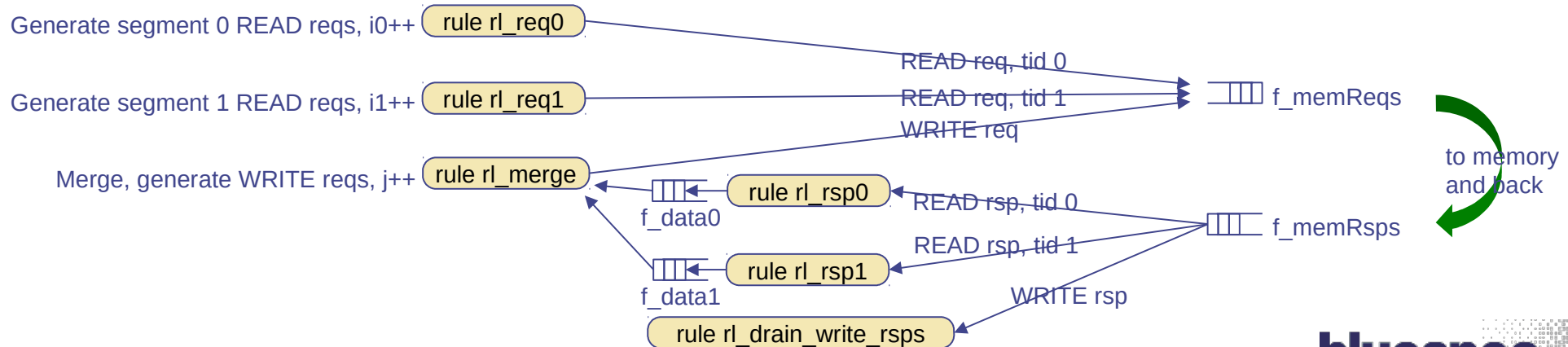
There is a danger of deadlock. Example:

- 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.

- rl_req0 and rl_rsp0 decrement and increment ehr_credits0, respectively, to never allow more than 8 requests in flight. rl_req1 and rl_rsp1 similarly maintain ehr_credits1.

This prevents the above deadlock situation.

mkMergeEngine module data flow

Generate segment 0 READ reqs, i0++ — rule rl_req0

Generate segment 1 READ reqs, i1++ — rule rl_req1

Merge, generate WRITE reqs, j++ — rule rl_merge

READ req, tid 0
READ req, tid 1
WRITE req

f_memReqs

to memory and back

rule rl_rsp0
f_data0
READ rsp, tid 0

rule rl_rsp1
f_data1
READ rsp, tid 1

f_memRsps

rule rl_drain_write_rsps
WRITE rsp

bluespec

In Resources/Memory_Model.bs: a memory model

To test our mergesort block, we need to provide it 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;
    interface Vector #(N_Mem_Ports, Server #(Req_T, Rsp_T)) bus_ifc;
    …
endinterface

module mkMemory_Model (Memory_IFC);
    …
endmodule
```

The interface provides N_Mem_Ports servers for memory requests. In Eg06a we will only use 1 port, but in Eg06c we will increase this, to model a memory with higher bandwidth.

The body of the module mkMemory_Model is fairly straightforward. It illustrates how we can import a C function to do some work (in simulation only). The associated files are also in Resources/:

- C_imports.{h,c}   C functions to 'malloc' an array representing memory, and to read/write the array
- C_import_decls.bs   BSV declarations to connect BSV to the C functions

bluespec

A word about "objdump" files

Our memory model initializes memory by reading in a file called "objdump".

Objdump files are a standard format on several flavors of Unix (including Linux and OS X)

Source code file
(asm, C, C++, …)

Assemblers, compilers
(as, gcc, clang …)

ELF executable file

GNU objdump utility

objdump file (text)

Disassembly of section .text
    … *program (machine code)* …

Disassembly of section .rodata
    … *read-only data* …

Disassembly of section .data
    … *data* …

Disassembly of section .bss
    …

… *other sections* …
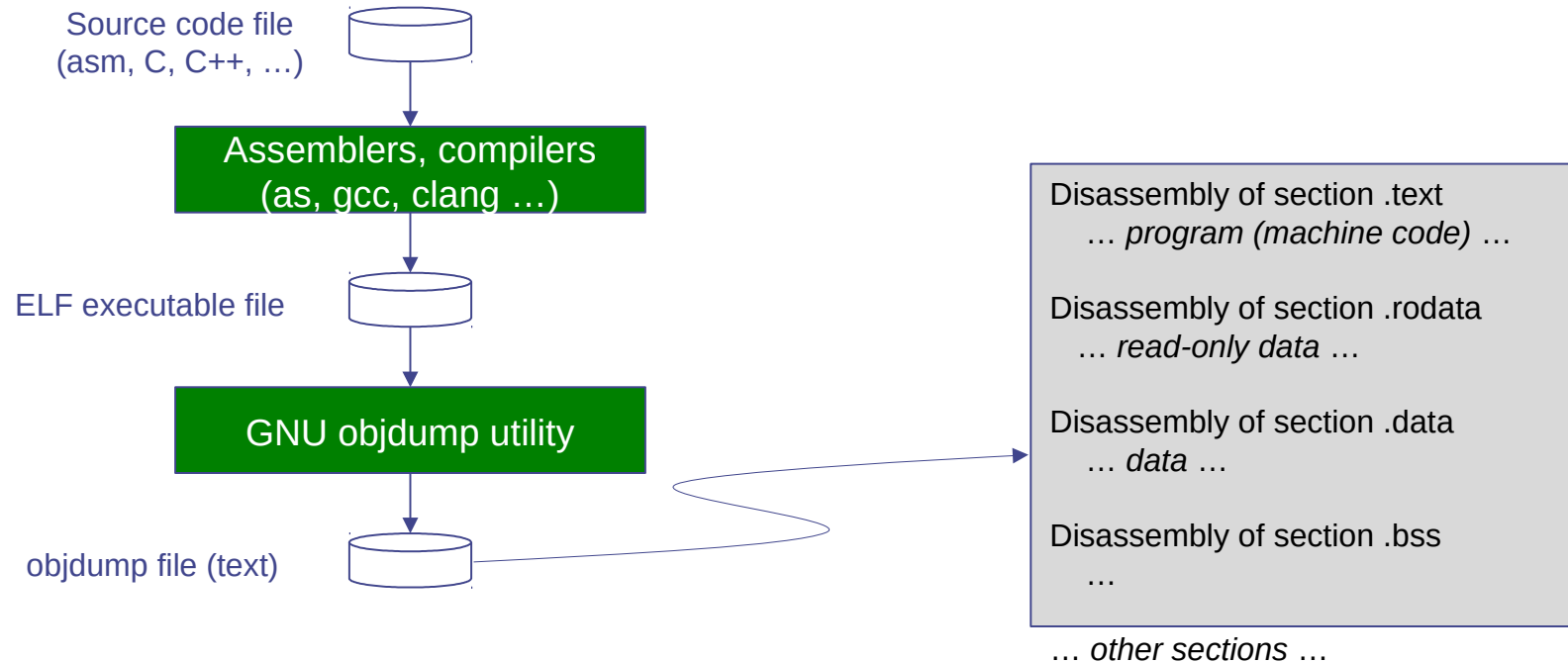
Note: We have not included any tools to create objdump files. The standard way is to use GNU tools. The pre-built objdump files distributed with this training were created using some ad hoc tools at Bluespec.

bluespec

In Testbench.bs: a testbench

Our testbench is excerpted below:

```
module mkTestbench (Empty) ;

   Memory_IFC     mem                     <- mkMemory_Model;
   Mergesort_IFC  mergesort               <- mkMergesort;

   mkConnection (mergesort.mem_bus_ifc, mem.bus_ifc [0]);
   …
   mkAutoFSM (
      seq
         mem.initialize (mem_base_addr, mem_size, init_from_file);
         mergesort.reset (accel_base_addr);
         dump_mem_range;
         … write mergesort's config regs …
         … loop, polling mergesort's config reg for completion …
         dump_mem_range;
      endseq);
endmodule
```

The testbench only performs a very small test so that you can easily inspect the outputs:

- Addr of the data array: 0x1000; scratch array: 0x1800; number of elements: 13

The 'dump_mem_range' calls (above) show memory contents before and after the sort.

(You are free to edit the program to try larger examples.)

bluespec

Build and run the 1st version

- In the Build directory, 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)

bluespec

In this version we use exactly the same Mergesort.bs as in Eg06a.

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

Please study: src/Sys_Configs.bs
in this directory, to see the changes to describe this new SoC.
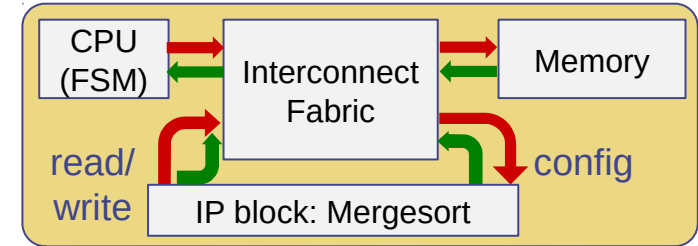
Note that we do some "type-level" arithmetic to derive the number of fabric targets based on the number of memory ports; to derive "initiator numbers" (INums) based on the number of initiators, etc.:

```
typedef  1  N_Mem_Ports;
typedef  2  Max_Initiators;     // CPU Data access, Accelerator data port

typedef  TAdd #(N_Mem_Ports, 1)  Max_Targets;

typedef  TLog #(Max_Initiators)  INum_Sz;

typedef  Bit #(TLog #(Max_Initiators))  INum;
typedef  Bit #(TLog #(Max_Targets))     TNum;
```
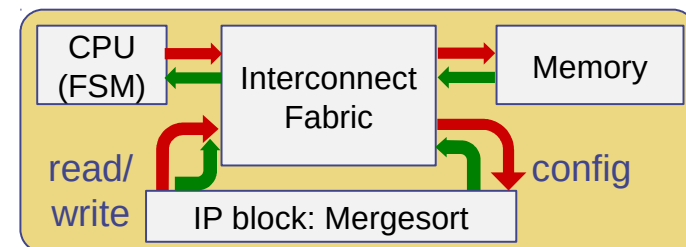
bluespec

Please study: src/Sys_Configs.bs
in this directory, to see the changes to describe this new SoC.

Since we have more than one initiator on the fabric, transaction ids for targets have log(N) more bits than transaction ids for initiators, where N is the number of initiators, because the fabric must tack on log(N) bits to remember which initiator must get the corresponding response:
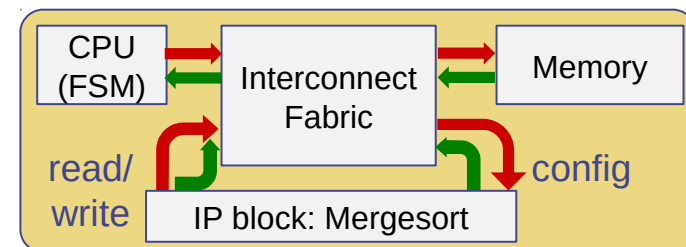
```
typedef  1  TID_SZ_I;

// Transaction ids at targets
typedef  TAdd #(TLog #(Max_Initiators), TID_SZ_I)  TID_SZ_T;

typedef Bit #(TID_SZ_I)  TID_I;
typedef Bit #(TID_SZ_T)  TID_T;
```

Finally, the end of the file now contains an "address decoder" module, which will be used by the Fabric to route memory requests either to the Memory or to the IP block, depending on the address.

bluespec

Please study: src/CPU.bs

You will see that it is not really a CPU, but just the testbench FSM from Eg06a, wrapped in a module that pretends to be a CPU. The interface is a step towards a real CPU interface, containing a "data-cache" interface, and other methods that will enable the GDB debugger to control the CPU:

```
interface CPU_IFC;
    // Interface to Data Memory
    interface Client #(Req_I, Rsp_I)  dcache_ifc;

    // GDB handling
    method Action           run_continue ();
    method CPU_Stop_Reason  stop_reason;
    method Action           req_read_memW (Addr addr);
    method ActionValue #(Data) rsp_read_memW ();
    method Action           write_memW (Addr addr, Data d);
endinterface
```

bluespec
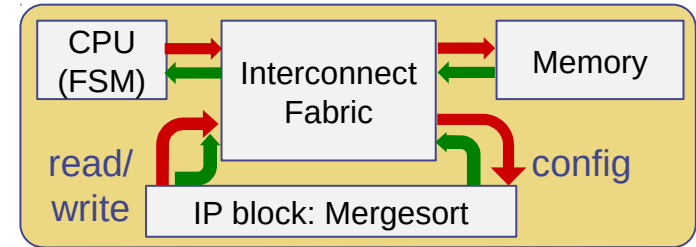
Please study: Resources/Fabric.bs

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

```
interface Fabric_IFC;
    interface Vector #(Max_Initiators, Server #(Req_I, Rsp_I))  v_servers;
    interface Vector #(Max_Targets,    Client #(Req_T, Rsp_T))  v_clients;
endinterface
```

Recall that Req_I is different from Req_T, and Rsp_I is different from Rsp_T:

- For requests, the fabric will tack on extra transaction id (tid) bits to remember the "return address" where responses should go
- For responses, the fabric will use those extra tid bits to route the responses, and will also strip them off to restore the original tid.

The module mkFabric is quite straightforward. It represents a "full crossbar" switch, i.e., there is a separate datapath from each input port to each output port, in both directions. These are represented by the rules that are generated in for-loops. The only interesting thing in each rule is the tid manipulation as described above.

24

bluespec

Please study: src/Testbench.bs

The system is instantiated with this excerpt:

```
module mkTestbench (Empty) ;

    CPU_IFC        cpu        <- mkCPU_Model;
    Memory_IFC     mem        <- mkMemory_Model;
    Fabric_IFC     fabric     <- mkFabric;
    Mergesort_IFC  mergesort <- mkMergesort;

    mkConnection (cpu.dcache_ifc, fabric.v_servers [cpu_d_iNum]);
    mkConnection (mergesort.mem_bus_ifc, fabric.v_servers [accel_iNum]);
    mkConnection (fabric.v_clients [mem_tNum], mem.bus_ifc [0]);
    mkConnection (fabric.v_clients [accel_tNum], mergesort.config_bus_ifc);
```
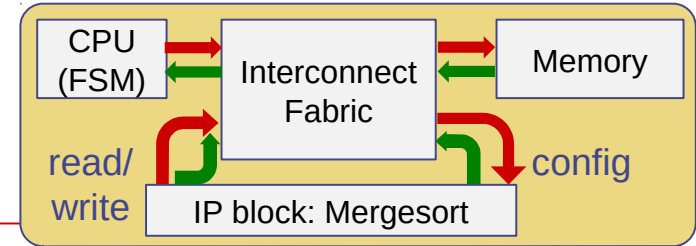
The behavior of the testbench (using mkAutoFSM) is a step towards a GDB-like interactive console to control the CPU. It uses an imported C command "c_console_get_command" to prompt the user for a GDB-like command and to return the command entered by the user, and then it executes the command.

In this first version, it recognizes just three commands: "continue" (to execute the program), "quit", and "memory dump" to show a region of memory.

Diagram (top right):
CPU (FSM) ↔ Interconnect Fabric ↔ Memory
read/write   config
IP block: Mergesort

bluespec

Build and run the 2nd version

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

- When you simulate
  - You will initially see initially see some INFO messages from the memory model, loading the objdump file
  - You will see the output of the first 'dumpmem', showing memory contents before the sort
  - Then, you will get a GDB-like prompt:

    ```
    Command? [type 'h' for help]:
    ```

  - Go ahead and type 'h' for a list of commands. It will list several GDB-like commands, but In this first version, it recognizes just three commands: "continue" (to execute the program), "quit", and "memory dump" to show a region of memory.
  - Type 'c' for 'continue', and it will perform the mergesort and do the final 'dumpmem' showing the memory contents after sorting. You can also use the 'm' command to display this memory region.
  - Type 'q' to quit back to the terminal prompt.

bluespec

In this version we use the same source codes for the SoC environment (although we will increase the number of initiators and targets on the Fabric).

We generalize the Mergesort module to have multiple merge engines (instead of just one) that can operate in parallel. Each merge engine will have its own initiator port on the interconnect fabric. The source code is parameterized to have N_Mergers; in the example code we instantiate this to 2.

Correspondingly, we also increase the number of target ports for the memory. The source code is parameterized to have N_Mem_Ports; in the example code we set that to 2.
Please study: src/Sys_Configs.bs
in this directory, to see the changes to describe this new SoC.
The overall structure is the same; just the details have changed to describe the extra target and initiator ports.
The address-decode function now interleaves memory addresses across the memory ports in 8-byte steps (addr 0..7 in first port, 8..15 in second port, … and so on).

The file: src/CPU.bs
is unchanged.

CPU (FSM) — Interconnect Fabric — Memory
read/write — IP block: Mergesort — config

bluespec
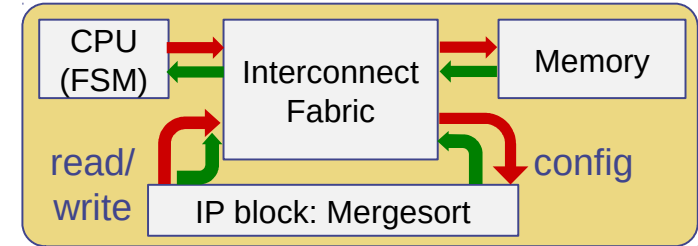
Please study: src/Mergesort.bs

It's memory interface has now become a vector of Clients:

```
interface Mergesort_IFC;
    method Action reset (Addr base_addr);
    interface Server #(Req_T, Rsp_T) config_bus_ifc;
    interface Vector #(N_Mergers, Client #(Req_I, Rsp_I))  mem_bus_ifc;
endinterface
```
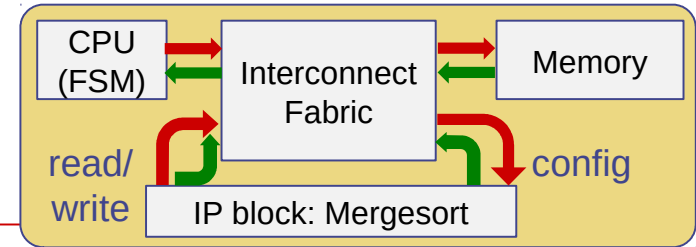
In the module mkMergesort, we now instantiate a vector of merge engines, instead of just one:

```
 Vector #(N_Mergers, MergeEngine_IFC) mergeEngines <- replicateM (mkMergeEngine);
```

In the module mkMergesort's behavior, where we used to start the single merge engine:

```
mergeEngine.start (0, 0, vrg_configs [n], rg_p1, rg_p2, vrg_configs [n]);
```

instead, we now just enqueue another "task" on to a queue:

```
f_tasks.enq (tuple4 (0, vrg_configs [n], rg_p1, rg_p2));
```

and, concurrently, we feed these tasks to any available merge engine:

```
for (Integer j = 0; j < valueOf (N_Mergers); j = j + 1)
    rule rl_exec_task;
        match { .i, .span, .p1, .p2 } = f_tasks.first; f_tasks.deq;
        mergeEngines[j].start (fromInteger (j), i, span, p1, p2, vrg_configs [n]);
    endrule
```

bluespec

Memory ordering problem
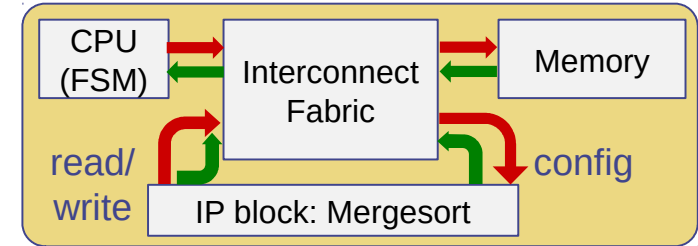
When we introduce multiple memory ports with interleaved addresses in an SoC, we introduce a memory ordering problem:
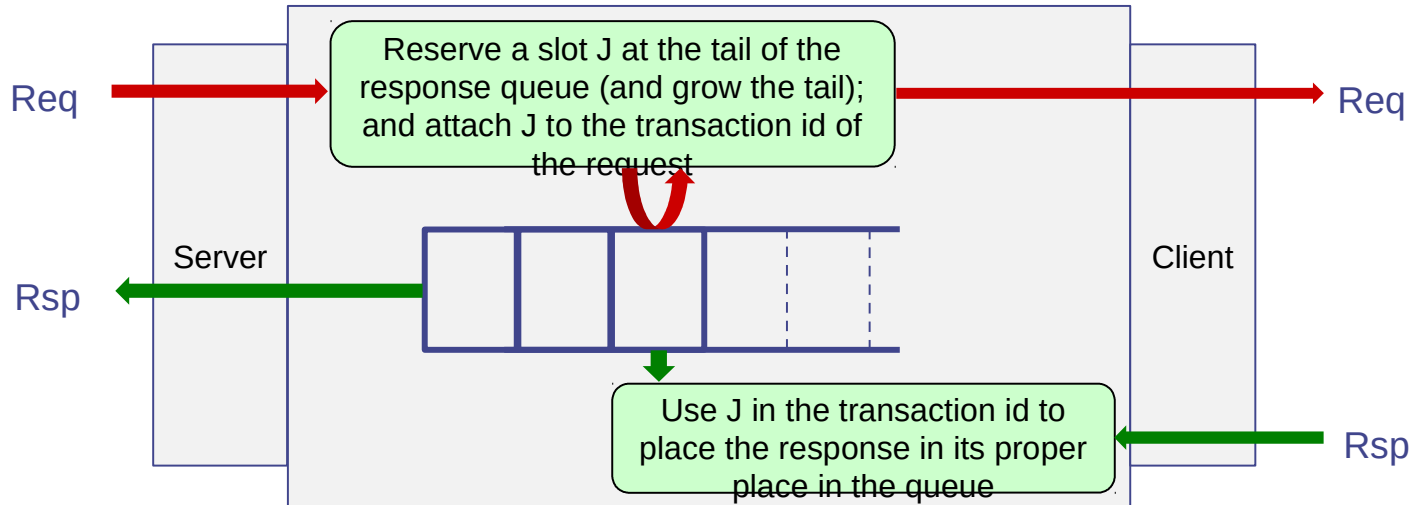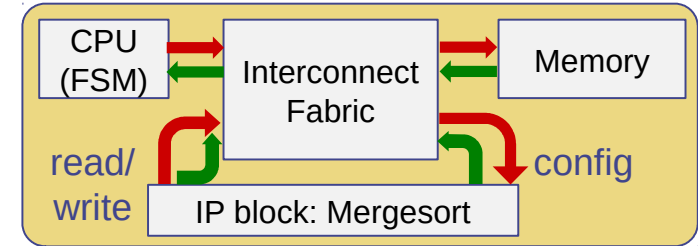
- Suppose the IP block sends two requests, one after the other, to memory at two addresses Addr1 and Addr2.
- These requests may go to two different memory ports, depending on which port services which address
- Since those two ports may be access different regions of memory and face different delays and contention, the responses may come back in a different order
- If the IP block assumes that responses come back in the same order as requests, it will produce wrong results!

Our mkMergeSort module does assume that responses will come back in order!
(Exercise: study the code and convince yourself that it makes such an assumption.)

Solution: we place a "reorder buffer" between the IP block and the Fabric, to restore the order of responses delivered from the Fabric to the IP block.

bluespec

3rd version: directory   Eg06c_Mergesort/   (contd.)

Please study: Resources/Reorder_Buffer.bs

- An incoming request reserves a slot J at the current tail of the response queue (and grows the tail). The position J is carried along with the request in its transaction id.
- A response is inserted into its correct position in the response queue by looking at J in the transaction id

Req

Reserve a slot J at the tail of the response queue (and grow the tail); and attach J to the transaction id of the request

Req

Server

Client

Rsp

Use J in the transaction id to place the response in its proper place in the queue

Rsp

30

© Bluespec, Inc., 2013-2017

bluespec

Please study: src/Testbench.bs

The system is instantiated with this excerpt:

```
module mkTestbench (Empty) ;

    CPU_IFC         cpu       <- mkCPU_Model;
    Memory_IFC      mem       <- mkMemory_Model;
    Fabric_IFC      fabric    <- mkFabric;
    Mergesort_IFC   mergesort <- mkMergesort;

    mkConnection (cpu.dcache_ifc, fabric.v_servers [cpu_d_iNum]);

    for (Integer j = 0; j < valueOf (N_Accel_Clients); j = j + 1) begin
        Reorder_Buffer_IFC reorder_buffer <- mkReorder_Buffer;
        mkConnection (mergesort.mem_bus_ifc [j], reorder_buffer.server);
        mkConnection (reorder_buffer.client, fabric.v_servers [accel_iNums[j]]);
    end

    for (Integer j = 0; j < valueOf (N_Mem_Ports); j = j + 1)
        mkConnection (fabric.v_clients [mem_tNums [j]], mem.bus_ifc [j]);

    mkConnection (fabric.v_clients [accel_tNum], mergesort.config_bus_ifc);
```
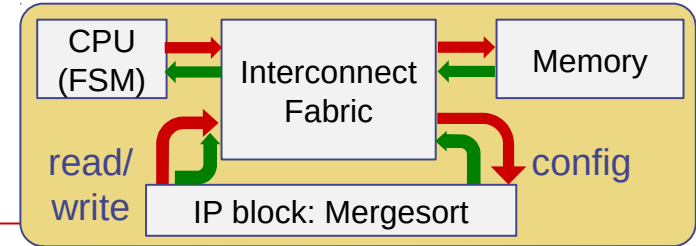
i.e., the for-loops connect the vector of mergesort initiators and memory targets to the corresponding ports of the interconnect fabric. We also place reorder buffers on the mergesort initiator ports.

bluespec

Build and run the 3rd version

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

- When you simulate you will see the same GDB-like command prompt as in Eg06b. Type 'c' (continue) to run the mergesort, and verify that the output looks reasonable.

bluespec

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 $\equiv$ the size is fixed at compile time). Note that Resources/Req_Rsp.bs already defines an enum type TLMBSize 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

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)

bluespec

End