Gorilla v.2: A Compiler for Generating Data-stream Accelerators

Maysam Lavasani

Jun 11, 2014

Contents

1 Introduction . . . 2
2 Improvements from Gorilla v.1 . . . 2
3 Gorilla language . . . 3
  3.1 Case study: A layer-3 header processor . . . 3
  3.2 Interface specification . . . 6
  3.3 Processing steps . . . 7
  3.4 Data-types . . . 7
  3.5 Predication . . . 8
  3.6 Transitioning between processing steps . . . 8
  3.7 Input/output operations . . . 8
  3.8 Engine language BNF . . . 8
4 Engine execution model . . . 10
  4.1 Latency-insensitive interfaces . . . 10
  4.2 Global state memory . . . 10
  4.3 Data-parallel execution . . . 10
5 Engine templates . . . 11
  5.1 Simple state machine engines . . . 11
  5.2 Multi-threaded engines . . . 12
    Execution pipeline . . . 12
    Thread contexts . . . 13
    Receive/send processing steps . . . 13
    Offload dispatch and wakeup logic . . . 13
  5.3 Pipelined engines . . . 14
    Pipeline registers . . . 16
    Receive/send stages . . . 16
    Split-phase stages . . . 16
    Stall logic . . . 16
6 Conclusion . . . 16

1 Introduction

Gorilla is a compiler for generating high performance accelerators that receive incoming streams of data, process them, and generate outgoing streams of data. The first version of Gorilla was developed as part of the research in [5]. We enhanced Gorilla features and developed Gorilla v.2 as part of "programmable accelerators" project. This document covers the language specification of Gorilla and, also, elaborates on Gorilla hardware compilation process.

A Gorilla accelerator consists of one or more processing engine(s) which correspond to different stage(s) of processing a data-element in the incoming streams of data (See section 3.1 for an example). Each engine is programmed in a stylized-C language which gives the impression of sequential programming to the designer. Gorilla compiler generates a synthesizable hardware model from the input code for each engine. It can implement two different concurrency mechanisms, pipelining and multi-treading, in the engine's micro-architecture. The concurrency mechanisms improve the quality of the generated accelerator.

2 Improvements from Gorilla v.1

The main improvements in Gorilla v.2 comparing to the first version is due to the change of compiler's output language. Gorilla v.1 translates the input program into Verilog code. Gorilla v.2, however, translates the input program into Chisel. Chisel [1] is a hardware construction language with both object-oriented and functional features. The generated Chisel code, later, can be fed to the Chisel compiler that can, in turn, generates either a synthesizable Verilog model or a cycle-accurate C-simulation model.

The new features in Gorilla v.2 include:

- Cycle-accurate C-simulation, inherited from Chisel C-simulation
- Bundle types, inherited from Chisel bundles (see section3.4)
- Split-phase pipelining (see section 5.3)
- Composing automatic generated engines using Chisel code

3 Gorilla language

The language for programming an engine is to model a state machine consisting of several processing steps [1].

The input to the Gorilla compiler can be generated by a high-level synthesis (HLS) tool from standard C [2]. Alternatively, when the quality of scheduling is critical for achieving a high performance, the application can be scheduled manually and written directly in form of Gorilla input language.

In this section, we use a layer-3 header processor as a case study to describe the Gorilla language. We, also, discuss individual features and present the Backus-Naur Form (BNF) of the Gorilla language.

3.1 Case study: A layer-3 header processor

Figure 2 shows a packet processor accelerator which (i) recieves an stream of incoming packets, (ii) determines the packets destination ports based on the result of lookup process, and (iii) sends the outgoing packets to the output ports.

The accelerator consists of five main engines chained in a back-to-back fashion. The first engine (i) splits the header and payload of the packet, (ii) sends the header to the output, and (iii) stores the payload in a payload memory. The second engine assigns a sequencing tag to each header. The third engine processes the header and determines its destination address. The fourth engine reorders the headers based on their sequencing tag. This is necessary if header processor processes multiple headers in parallel and cause the headers to get out of their incoming order. The last engine reassembles the packet from its header and the body that is stored in the payload memory. Lookup and QoS counting engines are used by header processor as specialized functional units.

We focus on programming one of the engine in this architecture, the header processor, using gorilla language. Figure 1 shows the high-level flow-chart of the packet processing engine. After checking layer-2 and layer-3 headers, the destination port is looked up. The engine updates QoS counters to keep track of routed packets to a particular destination port. Finally, the engine updates the TTL field in the packet and, also, updates the checksum field accordingly. If there is an error in any of these processing steps, the packet is marked as an exceptional

[1]The states in the program are different from actual hardware states. Each state in the engine program may be translated into one or two hardware state(s) or one or two hardware pipeline stages (See section 5). To avoid confusion, we refer to the states in the input state machine as processing steps

[2]Currently, gorilla is not integrated with any HLS tool. However, our initial study indicates that the output of LegUp [2] scheduler is close to Gorilla input language

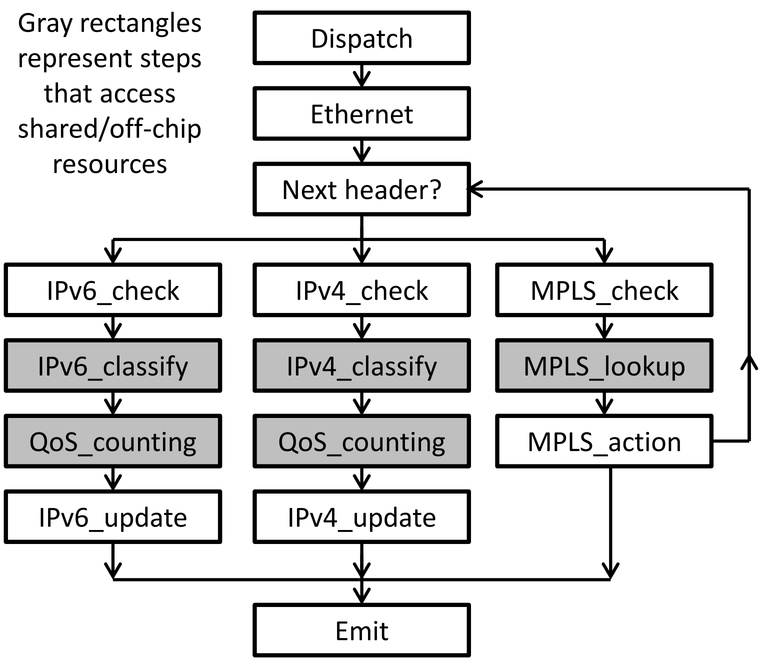

Figure 1: IPv4 header processing flow-chart

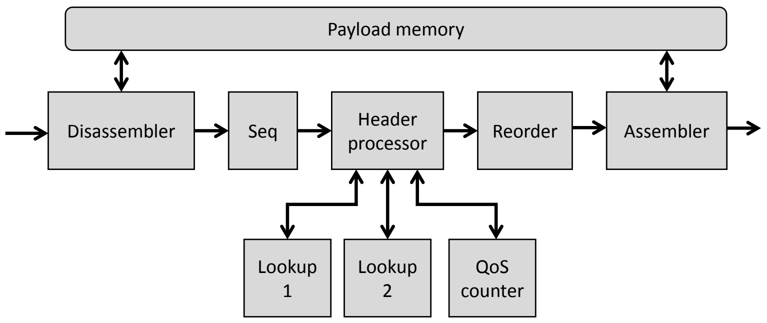

Figure 2: A multi-engine accelerator for routing the packets. Each engine is generated using Gorilla compiler.

packet and sent to the control-plane [3] for further processing.

Following is the gorilla code for implementing the above discussed header processor engine. For the sake of simplicity, we only show the code associated with processing an IPv4 packet.

```
#pragma INPUT NP_EthMpl3Header_t
#pragma OUTPUT NP_EthMpl3Header_t
#pragma OFFLOAD(ipv4Lookup1, uint32_t, uint8_t)
#pragma OFFLOAD(ipv4Lookup2, uint32_t, uint8_t)
#pragma OFFLOAD(qosCount, uint32_t, uint8_t)
```

[3]In high-end Internet routers network processors are part of the router data-plane. The control-plane, however, depends on software on general-purpose hardware. One of the duties of the contorl-plane is handling exceptional packets

```
#pragma CONCURRENT_SAFE

typedef struct {
  uint4_t version, hLength;
  uint8_t tos;
  uint16_t length, identification, flagsOffset;
  uint8_t ttl, protocol;
  uint16_t chksum;
  uint32_t srcAddr, dstAddr;
} IPv4Header_t;
typedef struct {
  uint48_t dstAddr, srcAddr;
  uint8_t l3Type, length;
} EthernetHeader_t;
typedef uint1024_t mpl3Header_t;
typedef struct {
 uint8_t outPort;
 EthernetHeader_t  eth;
 mpl3Header_t  l3;
} NP_EthMpl3Header_t;
IPv4Header_t ipv4Input, ipv4Output;
uint32_t gOutPort;

GS_ETHERNET()
{
  ipv4Input = (IPv4Header_t) Input.l3;
  Output = Input;
  if (Input.l2Protocol == ETHERNET) {
    State = GS_IPV4;
  } else {
    State = GS_EXCEPTION;
  }
}
GS_IPV4_CHECK()
{
  if (Input.eth.l3Type == IPV4) {
    State = GS_LOOKUP;
    ipv4Output = ipv4Input;
  } else {
    State = GS_EXCEPTION;
  }
  if (ipv4Input.length < 20 || ipv4Input.version != 4) {
    State = GS_EXCEPTION;
  }
}
GS_IPV4_CLASSIFY()
```

```
{
  uint outPort, srcLookupResult;
  outPort = ipv4Lookup1(ipv4Input.dstAddr);
  srcLookupResult = ipv4Lookup2(ipv4Input.srcAddr);
  Output.outPort = outPort;
  gOutPort = outPort;
  if (srcLookupResult == INVALID_ADDRESS ||
   outPort == INVALID_ADDRESS) {
    State = GS_EXCEPTION;
  } else {
    State = GS_UPDATE;
  }
}
GS_QOS_COUNTING()
{
  uint8_t qcOutput;
  qcOutput = qosCount(gOutPort);
}
GS_IPv4_UPDATE()
{
  if (ipv4Input.ttl == 1) {
    State = GS_EXCEPTION;
  } else {
    ipv4Output.ttl = ipv4Input.ttl - 1;
    ipv4Output.chksum = ipv4Input.chksum + 0x80;
  }
  Output.l3 = (mpl3Header_t) ipv4Output;
  finish();
}
GS_EXCEPTION()
{
  Output.outPort = CONTROL_PLANE;
  finish();
}
```

3.2 Interface specification

Each engine has one input, one output, and a set of offload interfaces. The input and output are one-way interfaces. Offloads are two-way interfaces for sending requests and receiving the responses. The engine interface types are specified using C pragmas at the beginning of an engine code.

Both the input and output of the header processing engine are defined as a bundled type which consists of Ethernet header and a generic type for a layer-3 header. The bundle, also, consists of an output port to specify the result of lookup process [4].

The header processor have three offload interfaces. Two interfaces are used to connect to lookup engines and one interface is used to connect to QoS counter engine. Each offload interface consists of three parts. The Offload name, the input type, and the output type. For all of these three offload interfaces, the input types are 32 bits and the output types are eight bits.

3.3 Processing steps

Each processing step is defined using a C function and can perform arbitrary arithmetic/logic computation. The state can, also, call special purpose functional units. The calls are, later, translated to request/response messages to/from the functional units through the engine's offload interfaces. Each state, also, explicitly specifies the next state(s) based on the result of computation and/or the responses form functional units. Note that the functional units, themselves, are processing engines and can be generated by Gorilla compiler.

The packet processing engine consists of six processing steps. The lookup processing step calls two different lookup functional units in parallel. The update processing step calls the QoS counter functional unit.

3.4 Data-types

Apart from standard C data-types, Gorilla supports bit accurate data types. The bit accurate data-types can be specified in forms of predefined uint data-types. For example, uint48_t specifies a 48 bits unsigned integer.

Bundled types are specified using C structures. Gorilla translates C structures are translated into Chisel bundles. The support of bundled data-types can be used for processing protocol fields. The header processor code uses bundled types for specifying Ethernet and IPv4 protocols.

Type casting can be used to convert a variable with a given data-type to another data-type. Casting a bundle type to another results into flattening the source bundle into a bit stream and remapping the bit stream into the target bundle type. The header processor code uses type casting to convert a generic layer three header to an IPv4 header bundle. The programmer, later, can access the IPv4 protocol fields as the fields of a C structure.

[4]The output port field in the input interface is not used. The same type is used for the input as the output for the sake of simplicity

3.5 Predication

The programmer can use conditional statements in each processing step. The conditional statements are translated into predication logic by the compiler. The header processor, for example, uses if-statements to check whether there is an error in processing the header.

3.6 Transitioning between processing steps

Transitioning between processing steps are done by explicitly writing to **State** variable. At the end of each processing step, the Gorilla infrastructure switches to the state that is specified by this variable.

3.7 Input/output operations

The input data element is accessible through **Input** variable. Output should be written to **Output** variable. Receiving an input is done implicitly when engine restarts in the first processing step. Sending an output is done either by calling **finish** routine or **emit** routine. The difference between finish and emit is that finish returns to the first processing step of the engine but emit returns back to the processing step specified as its argument. By calling the finish method, the header processor emits the output header and also returns back to the first processing step.

3.8 Engine language BNF

⟨*program*⟩ ::= ⟨*interface-pragmas*⟩ ⟨*type-defintions*⟩ ⟨*variable-defintions*⟩ ⟨*processing-steps*⟩
  | ⟨*type-defintons*⟩

⟨*interface-definitions*⟩ ::= ⟨*input-defintion*⟩ ⟨*output-defintion*⟩ ⟨*offload-defintions*⟩

⟨*input-definition*⟩ ::= ‘`#pragma`’ ‘`INPUT`’ ‘`(`’ ⟨*type-ident*⟩ ‘`)`’

⟨*output-definition*⟩ ::= ‘`#pragma`’ ‘`OUTPUT`’ ‘`(`’ ⟨*type-ident*⟩ ‘`)`’

⟨*offload-defintions*⟩ ::= ⟨*offload-definition*⟩ ⟨*offload-definitions*⟩
  | ⟨*offload-definition*⟩

⟨*offload-definition*⟩ ::= ‘`#pragma`‘ ‘`OFFLOAD`’ ‘`(`’ ⟨*type-identifer*⟩ ‘`,`’ ⟨*offload-ident*⟩
    ‘`)`’

⟨*variable-defintions*⟩ ::= ⟨*variable-definition*⟩ ⟨*variable-definitions*⟩
  | ⟨*variable-definitions*⟩

```
⟨variable-definition⟩ ::= ⟨type-ident⟩ ⟨variable-list⟩

⟨varialbe-list⟩ ::= ⟨variable-id⟩ ⟨variable-list⟩
  |  ⟨variable-id⟩

⟨processing-steps⟩ ::= ⟨processing-step⟩
  |  ⟨processing-step⟩ ⟨processing-steps⟩

⟨statatement-list⟩ ::= ⟨statement⟩ ⟨statement-list⟩
  |  ⟨statement⟩

⟨processing-step⟩ ::= ⟨processing-step-name⟩ ‘( )’ ‘{’ ⟨variable-defintions⟩ ⟨statment-list⟩
     ⟨offload-statement-list⟩ ⟨statement-list⟩ ‘}’

⟨statement⟩ ::= ⟨ident⟩ ‘=’ ⟨expr⟩ ‘;’
  |  ⟨if-else-statement⟩
  |  ⟨if-statement⟩
  |  ‘{’ ⟨stattement-list⟩ ‘}’
  |  ⟨emit-statement⟩

⟨if-statement⟩ ::= ‘if’ ‘(’ ⟨expr⟩ ‘)’ ‘{’ ⟨statement-list⟩ ‘}’

⟨if-else-statemnt⟩ ::= if-statement ‘else‘ ‘{’ ⟨statement-list⟩ ‘}’

⟨offload-statement-list⟩ ::= ⟨offload-statement⟩ ⟨offload-statement-list⟩
  |  ⟨offload-statement⟩

⟨expr⟩ ::= ⟨ident⟩ ⟨binary-operator⟩ ⟨expr⟩
  |  ⟨ident⟩
  |  ⟨cast-prefix⟩ ⟨expr⟩
  |  ⟨ident⟩ ⟨field-expression⟩

⟨cast-prefix⟩ ::= ‘(’ ⟨type-identifier⟩ ’)’

⟨field-expression⟩ ::= ‘.’ ⟨ident⟩
  |  ‘.’ ⟨ident⟩ ⟨field-expression⟩

⟨offload-statement⟩ ::= ⟨ident⟩ ‘=‘ <offload-ident> ‘(’ ⟨expr⟩ ‘)’ ‘;’

⟨emit-statement⟩ ::= ‘finish’ ‘(’ ‘)’ ‘;’
  |  ‘emit’ ‘(’ ⟨processing-step-name⟩ ‘)’ ‘;’

⟨type-definitions⟩ ::= ⟨type-defintion⟩ ⟨type-defintions⟩
  |  ⟨type-defintion⟩
```

$\langle$*type-defintion*$\rangle$ ::= ‘`typedef`’ $\langle$*type-id-new*$\rangle$ $\langle$*type-id-old*$\rangle$
 | ‘`typedef`’ ‘`struct`’ ‘`{`’ $\langle$*variable-defnitions*$\rangle$ ‘`}`’ $\langle$*bundle-type-id*$\rangle$ ‘`;`’

4 Engine execution model

This section covers three important concepts associated with the engine execution model. The communication mechanism, parallelism opportunities, and handling the global state. Understanding these concepts is imperative for understanding the hardware that is generated by Gorilla compiler.

4.1 Latency-insensitive interfaces

Gorilla engines use latency-insensitive protocol [3] for communicating through input, output, and offload interfaces.For each way of an interface a valid signal from the source to the sink indicates whether the source has a data element to transfer. Also, a ready signal from the sink to the source indicates whether the sink side is ready to receive a data element. Transfer happens when both of these signals are asserted.

4.2 Global state memory

Gorilla engines cannot keep any internal data that is accessible by processing steps associated with two (or more) input data-element. Therefore, if we reset all engine's registers after processing a data-element, the outcome of the computation will be the same [5].

Gorilla engines can keep the global state, between processing of data-elements, in offloaded memory elements, however. The QoS counter in header processor example is a global state that is accessible by processing steps of all headers.

4.3 Data-parallel execution

We assume that there is no dependency between processing steps. As the result, the generated concurrency mechanisms in the hardware, either multi-threading or pipelining, does not give any priority between processing steps of different input data-elements.

[5]We are assuming that the engine has single thread. If it has more than one thread this argument will be change to: “after a thread finishes processing a data-element, resetting that particular thread's registers will not affect the outcome of the computation”

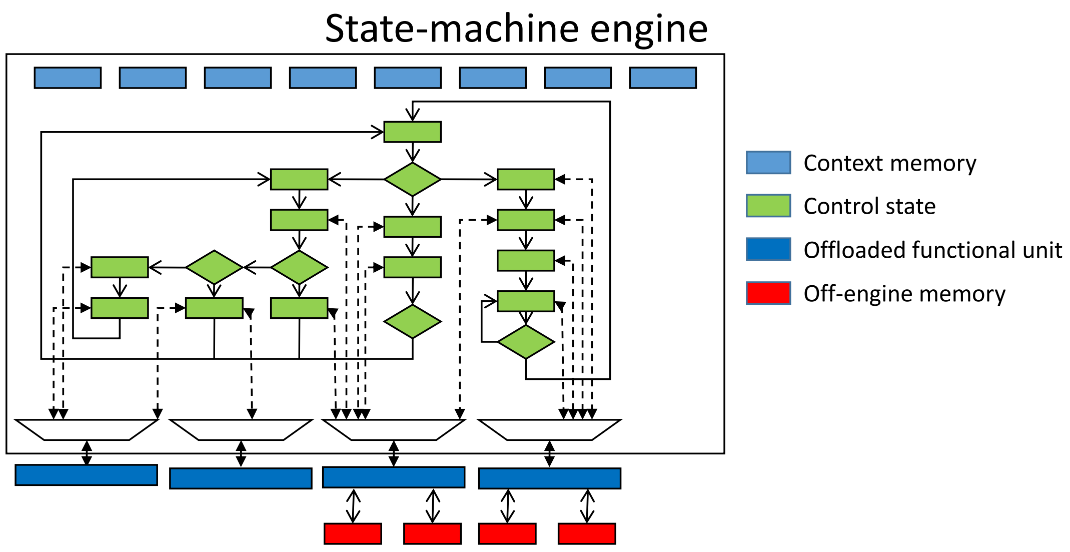

Figure 3: A simple state machine engine with

If any dependency exists between processing the input data-elements, the programmer should use synchronization mechanisms as offloaded functional units to enforce the priority in scheduling the threads or stalling the pipeline.

5 Engine templates

Gorilla compiler uses a template-based method to compiler processing steps into the hardware. Templates are generic and highly parametrized form of the hardware that will be specialized based on the functionality specified in input program. The template is designed for arbitrary number of functional units, arbitrary number of processing steps, and arbitrary data-types for interfaces.

Gorilla has three main templates that generate (i) simple state machine engines, (ii) multi-threaded engines, and (iii) pipelined engines.

5.1 Simple state machine engines

When an engine is translated into a single state machine all processing steps that does not have any offload calls are translated into a single hardware state and all processing steps that have offload calls are translated into two hardware states, a state for compuations before offload call(s), preOff state, and a state for compuations after offload call(s), postOff state. Global variables are translated into registers and local variables are translated into wires. Figure 3 shows an implemented engine using a simple state machine tempalte.

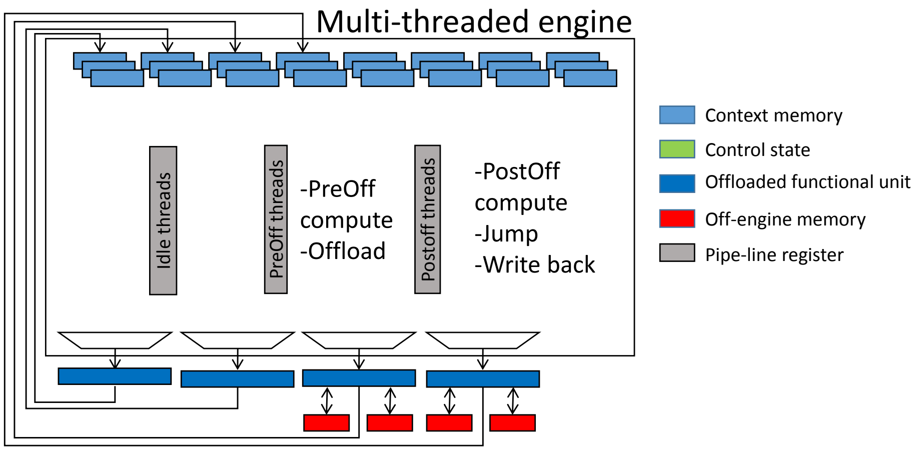

Figure 4: A multi-threaded engine

5.2 Multi-threaded engines

Since the target applications of Gorilla are data-parallel in nature, whenever an engine is spending a large amount of time in the offload call(s), it is reasonable to switch to the processing of another input data element while waiting for the response of the offload call(s). Inspired by multi-thrading technique in high-performance processors [4], we refer to this type of concurrency as multi-threading. Each thread in an engine is responsible for processing an input data-element. Whenever a thread calls an offload, the engine switches to another thread, processing another input data-element. Therefore, offload latencies in a thread are hidden by performing compuation in other threads.

Execution pipeline

A multi-threaded engine (shown in Figure 4) executes the processing steps using a two-stage execution pipeline. The first stage of the pipeline, also known as preOff stage, executes all the computations of a processing step before the offload calls. The second stage, also known as postOff stage, executes the computations after offload calls. The thread associated to a given data-element is either in preOff stage or postOff stage of the execution pipeline. Therefore, the preOff and postOff computations of a single input data element are not executed at the same clock cycle. Therefore, RAW, WAW, or WAR dependencies cannot occure for a single thread.

The engine keep track of threads using three bit-vectors. The first bit-vector indicates whether a thread is idle and waiting for an input data-element. The second bit-vector contains all threads that are in preOff stage, either performing the preOff computation or waiting for offload request acknowledgement. The third bit-vector contains all threads that are in postOff stage, either performing the postOff computation or waiting for offload replies.

Thread contexts

Thread contexts consist of all global variables in the input engine program. It, also, includes the implicit variables including Input, Output, State, and emitReturnState(See section 5.2) variables. Each of these variables is stored in an array which is indexed using thread id of the current active thread.

Receive/send processing steps

Each thread in a processing engine has an invisible processing step for receiving an input data-element. Also, it has an invisible processing step for sending an output data-element.

The free threads stays in receive processing step until an input data-element is assigned to them. When a new data-element arrives, a thread from the pool of available threads is selected and assigned for processing the data-element. The State variable for the thread is changed to the first processing step in the Gorilla program. The engine assert the ready signal on its input as long as there is an available free thread in the engine.

When threads emit an output, they automatically switch to the send processing step. An implicit variable keeps track of the step that the thread should switch to after successfully sending the output element. A thread stays in the send processing step until it receives the assertion of transfer (on the ready signal of the output port). While staying in this processing step the engine asserts its valid signal on the output port.

Offload dispatch and wakeup logic

Apart from the preOff computations specified in a processing step, the preOff stage is responsible for generating request(s) to offloaded functional units. The engine leaves the preOff stage and enters the postOff stage whenever all requests for all offload calls in a particular processing step are accepted by the corresponding functional units. Also, the engine leaves the postOff stage and enter the preOff stage of the next processing step whenever all replies form the offload calls are received [6].

[6]It is possible that more than one thread becomes available to move from postOff stage to preOff stage. An fair arbiter selects one of these threads at any given cycle.

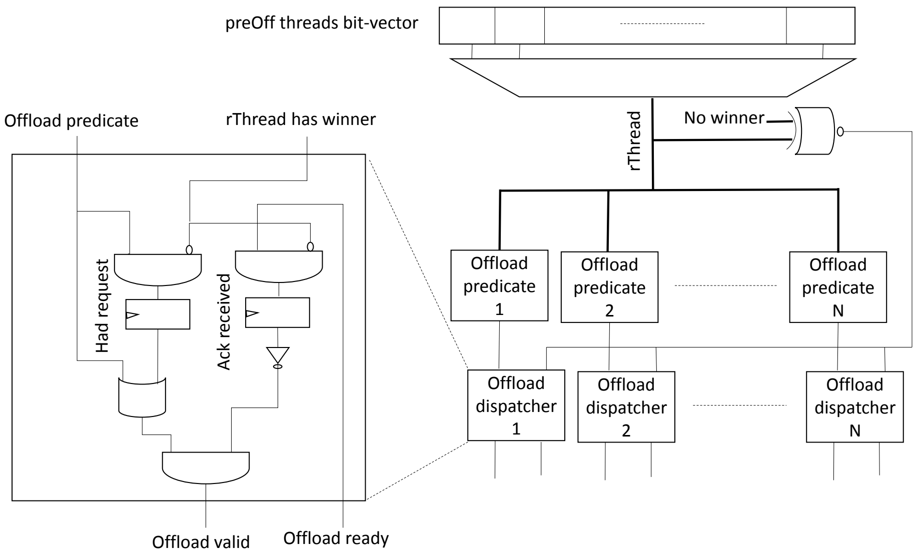

Figure 5: Offload dispatch logic

When a thread in a preOff stage calls an offloaded functional unit, a request is sent to the offloaded functional unit by asserting the valid signal on the corresponding request port. The request is asserted until the functional unit acknowledge receiving the request by asserting the ready signal on the same request port (See Figure 5 for details). The actual request data that is the argument of the offload call is derived along the request valid signal (Offload request bus is not shown in the Figure). The dispatch logic for a given functional unit can be asserted by any of the processing steps that call the functional unit. When last response from functional units of a given processing step receives, the corresponding thread is waken up and moved from postOff thread list to preOff thread list for processing the next Gorilla processing step . Figure shows the logic that determines if a given thread is done waiting for its offload calls in postOff stage.

5.3 Pipelined engines

Whenever the control flow between processing steps is solely jump from one state to its next step in the input program order, Gorilla can generate a pipelined engine.

Pipelining improves the throughput of the engine and can be useful if computation in the engine is a bottleneck in the design (multi-threading is useful when stalling on the offloaded functional units is a bottleneck in the design).

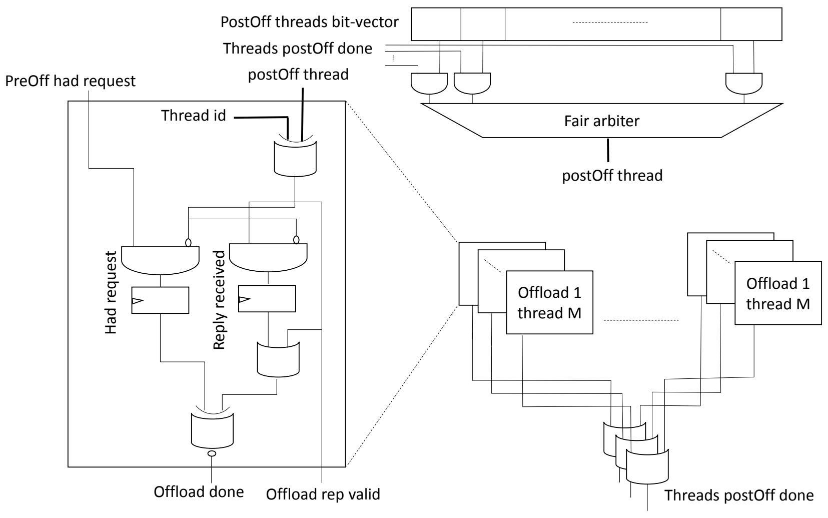

Figure 6: Offload thread wakeup logic

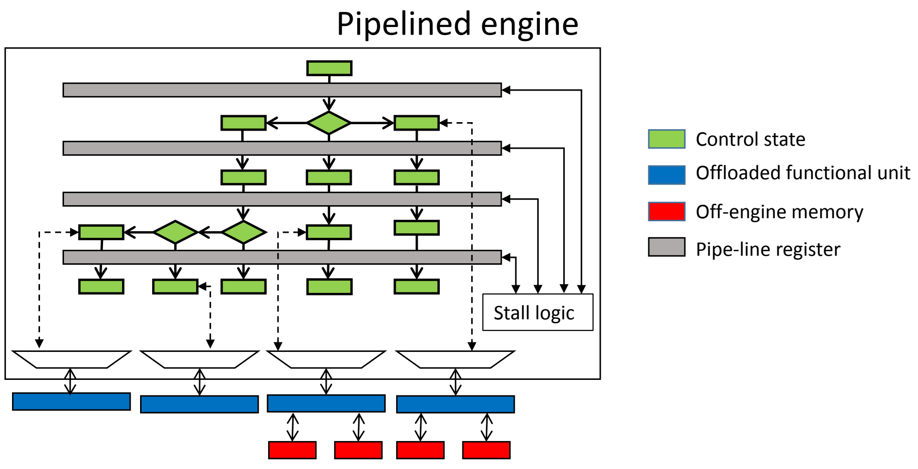

Figure 7: A pipeline engine for a loop-free input Gorilla program

Pipeline registers

Pipelined engine does not have any global context. The global variables are kept in pipeline registers. For a given input data element, corresponding global variables are marching along its pipeline stages.

Receive/send stages

A dedicated receive stage is added at the start of each Gorilla pipeline and a dedicated send stage is added at the end of each Gorilla pipeline.

Split-phase stages

Unlike other pipeline HLS tools, Gorilla code can call offload functional units and wait for the response of functional unit without stalling the whole pipeline. Gorilla compiler generates a preOff pipeline stage and a postOff pipeline stage for every processing step with a call to offloaded functional unit.

The structure of preOff dispatch logic and postOff wakeup logic in pipelined engine is very similar to the corresponding stages in a multi-threaded engine. The main difference is that unlike multi-threaded engine which uses the same preOff and postOff stages for executing all processing steps, a pipelined engine has dedicated preOff and postOff stages for each processing step with offload calls.

Stall logic

Each pipeline stage follows the latency-insensitive communication protocol which is described in 4.1. A preOff stage is stalled when there is not postOff thread available for accepting the result of preOff stage. When a preOff stage is stalled it ripple back the stall by de-asserting the ready signal. A postOff stage is stalled, however, whenever its next stage is stalled. The ready signal of the receive pipeline stage is connected to the whole engine ready signal and the valid signal of the send pipeline stage is connected to the whole engine valid signal.

6 Conclusion

This document described the details on the language specification and the hardware generation process of Gorilla compiler. Our primliminary experience with Gorilla show that high quality hardwre can be generated by combining engines that are generated by Gorilla. The concurrency mechanisms which are automatically generated by Gorilla is essentila in the quality of generated hardware. The combination of one-way input/output and two-way offload interafces are an expressive way to describe various composistion of interacting engines.

Bibliography

[1] J. Bachrach, Huy Vo, B. Richards, Yunsup Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic. Chisel: Constructing hardware in a scala embedded language. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 1212 –1221, june 2012.

[2] A. Canis, J. Chois, M. Aldham, V. Zhang, A. Kammoona, J. Anderson, S. Brown, and T. Czajkowski. Legup: high-level synthesis for fpga-based processor/accelerator systems. In *FPGA 2012*, FPGA '11, New York, NY, USA, 2011. ACM.

[3] L.P. Carloni, K.L. McMillan, and A.L. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 20(9):1059–1076, Sep 2001.

[4] Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, Rebecca L. Stamm, , and Dean M. Tullsen. Simultaneous Multithreading: A Foundation for Next-generation Processors. *IEEE Micro*, pages 12–18, September/October 1997.

[5] Maysam Lavasani, Larry Dennison, and Derek Chiou. Compiling High Throughput Network Processors. In *20th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, February 2012.