The Trimaran HPL-PD

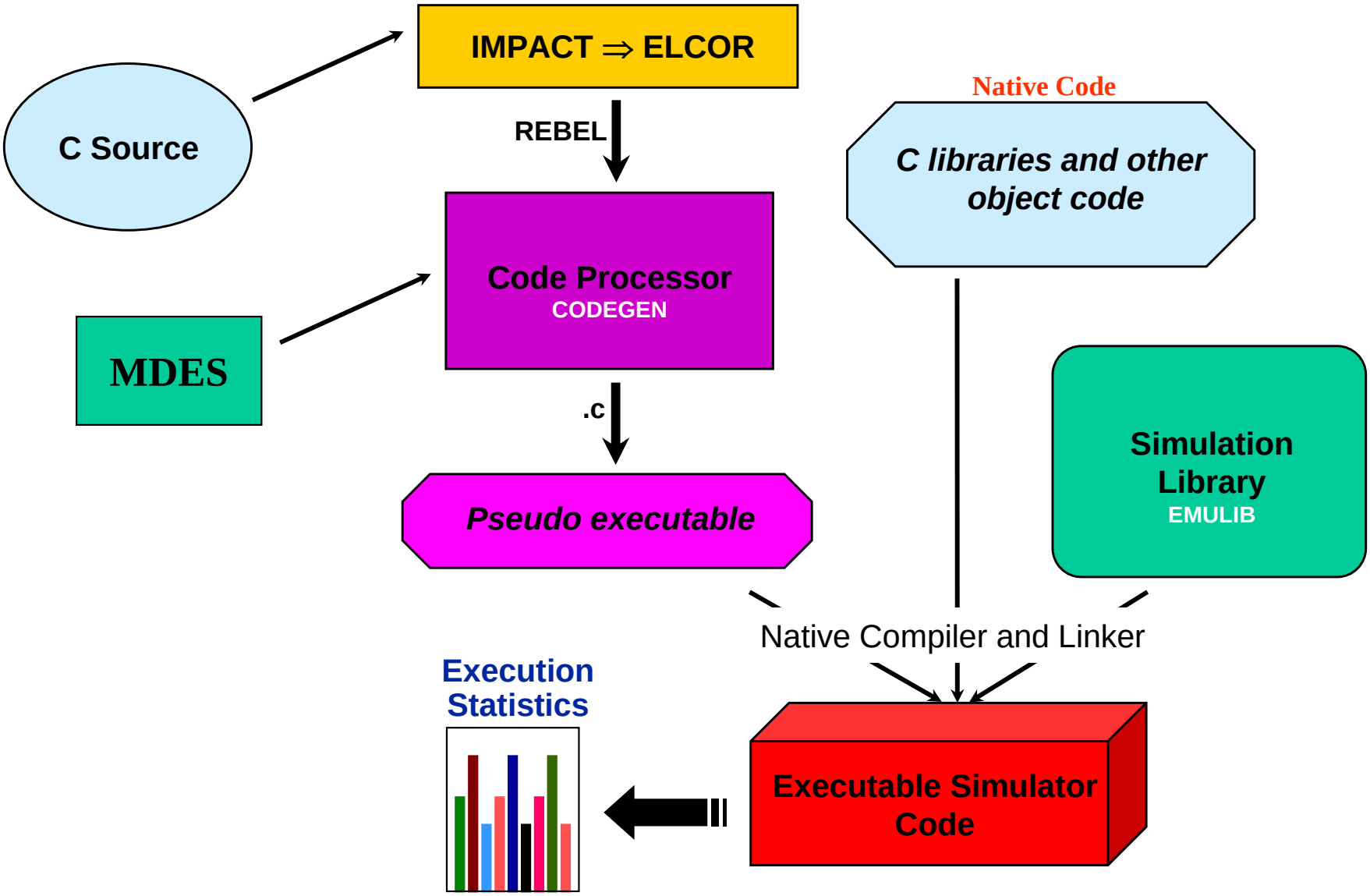
Simulation Infrastructure

Rodric Rabbah

The Framework

- The goals of the HPL-PD simulation framework are
  - Emulate the execution of the generated REBEL code on a virtual HPL-PD processor
  - Have the ability to adapt to changes in the machine description
  - Generate *accurate* run-time information
    - Execution clock cycles
    - Dynamic control flow and call trace
    - Address trace
    - Average number of operations executed per cycle

Overview


IMPACT ⇒ ELCOR
C Source
REBEL
Native Code
C libraries and other object code
Code Processor
CODEGEN
MDES
.c
Simulation Library
EMULIB
Pseudo executable
Native Compiler and Linker
Execution Statistics
Executable Simulator Code

General Description

- The code processor translates the REBEL input to a *C* equivalent
- The output of CODEGEN is a collection of *pseudo-executable* low-level *C* files
- A native compiler (i.e. GCC) is used to generate the equivalent machine code

General Description (continued)

- The pseudo-executables are compiled and linked with the *Simulation Library* and any other native libraries or object code
  - The generated executable is the simulator for the specified application

- The resultant executable is run on the host platform to generate statistics and dynamic profile information

EMULIB - The Simulation Back-End

The Simulation Library

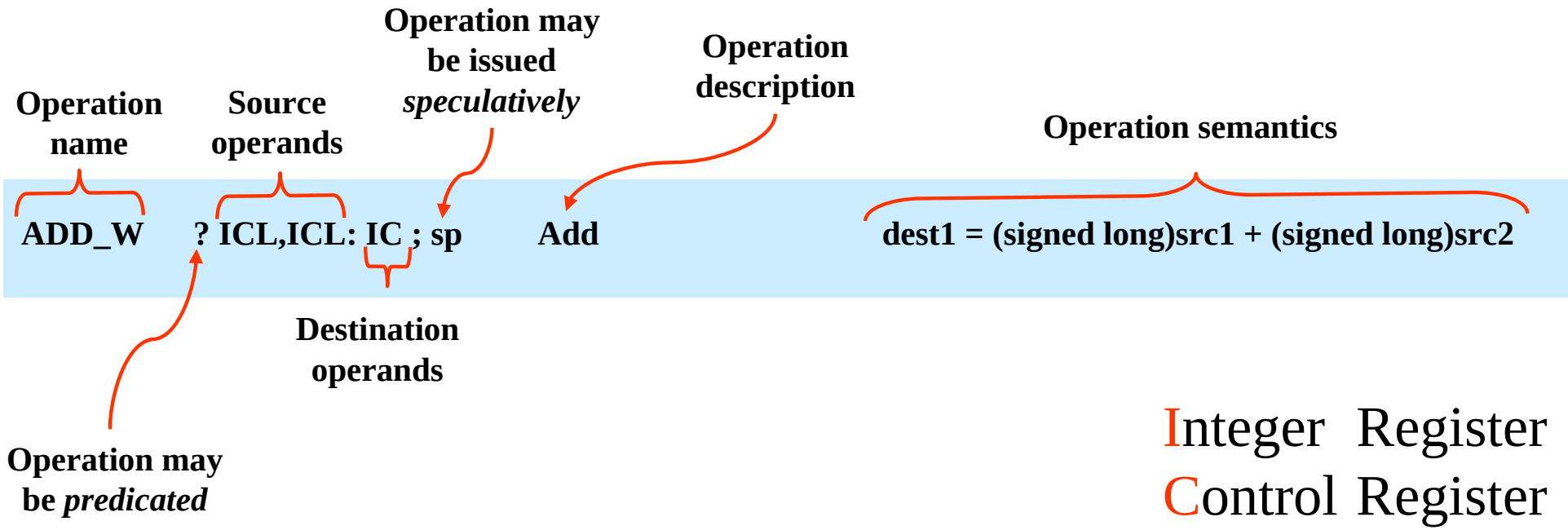
- EMULIB Consists of a *virtual machine interpreter* to execute HPL-PD operations
  - Supports EQUALS and LTE execution models
  - Supports speculative execution
- Accumulates various execution statistics
- Generates dynamic information
  - Control flow
  - Operation nullification
  - Address trace

The Simulation Library (continued)

- The virtual machine interpreter (VMI) is invoked on every procedure entry
  - VMI simulates the supplied operation stream in a loop until the procedure returns or the application terminates
- There is one function to simulate every HPL-PD operation
  - The simulation functions are automatically generated from a semantics description file

Simulation Function Generation

- A register transfer level type description of the operation format and its semantics are translated to a *C* function



Integer Register
Control Register
Literal

Function Generation (continued)

- Special *directives* can be used for special processing
  - \INIT #define DA_OC(_dest,_pred,_res) if ((_pred) && !(_res)) (_dest) = 1

    instructs the processor to copy the characters following the directive exactly as is
  - [xxx]

    instructs the processor to redirect the output to a file with name xxx
  - # This text is converted to a C style comment

    instructs the processor to convert the characters following the directive to a *C* style comment
  - \LOOP j X Y Z …

    instructs the processor to substitute each occurrence of $j with the X Y Z …

Function Generation (continued)

- *ops.list* defines all operations of the HPL-PD ISA

- *gen_functions* processes each operation defined in ops.list to automatically generate an equivalent *C* function
  - *A C* header file declaring the respective prototypes is also generated

HPL-PD Operation Simulation

- The generated functions compute the operation and save the result into local variables

- A computed value is not reflected in the processor state until *L* cycles after issue time have elapsed
  - *L* is the operation latency specified by the ELCOR compiler

Example Operation Simulation

```
function void PD_SUB_W_reg_lit(Op *op)
{
        REG r;
        r = (op → src[0] → reg) – (op → src[1] → lit);
        PD_delay(&r, &(op → dest[0] → reg), op → lat[0]);
}
```

- The assignment computes the subtract literal operation and saves the result in local variable *r*
- PD_delay schedules the update of the processor register (op → dest[0] → reg) to occur after (op → lat[0]) cycles

HPL-PD Op

- An operation in the simulation library is represented by a *C* structure PD_OP in PD.h

```
struct PD_OP {
   unsigned int      Rebel_id;              /* Rebel op id */
   PD_IMPL           op;                    /* pointer to simulation function */
   PD_PORT           pred;                  /* predicate port */

   PD_PORT           src [PD_MAX_SRC];          /* source ports */
   PD_PORT           dest[PD_MAX_DEST];         /* destination ports */
   unsigned int      lat [PD_MAX_DEST];         /* output latencies *

   /* debugging and statistics */
   PD_op_type        type;                  /* type of operation */
   unsigned int      Predicated;            /* Set to 1 when Predicated  Op */
   unsigned int      Speculated;            /* Set to 1 when Speculative Op */
   unsigned int      Mask;                  /* Set to 1 when Mask Op */
};
```

HPL-PD Port

- A PD_PORT is an operand
  - Literal and Register operands share the same *C* structure PD_PORT

```
struct PD_PORT {
   unsigned int  is_reg; /* register or literal operand */

   PD_REG        *file;  /* pointer to the register file or the actual literal value */
   unsigned int  num; /* register number when the operand is register */
   unsigned int  rot;    /* rotating register number when applicable */
};
```

HPL-PD Register

- The PD_REG *C* structure defines the representation of a register in the simulation library
  - One structure is used to represent floating point and integer values
- Each register has a *speculative tag* bit used to support speculative execution

The Simulation Loop

simplified interpreter loop from PD_main.c

```
function void PD_simulate(Op opsTable[])
{
        for (Op *opPtr = opsTable[0]; !done; opPtr++)
        {
                /* perform the operation */
                *(opPtr → op)(opPtr);
        }
}
```

Invokes the function which will simulate the current operation

Pointer to the current operation – contains source and destination operand information, latency information, and guarding predicate operand

The *Ops*Table

- The *Ops*Table is an array of PD_OPs
  - Generated by CODEGEN during the REBEL to *C* translation

```
PD_OP  opsTable_fir[75] = {
        {
            70,
            PD_SUB_W_lit_reg,
            { {(PD_REG *) 32,0,0x0,0}, {(PD_REG *) PD_GPR_FILE,4,0x0,1}, {0x0,0,0x0,0}, {0x0,0,0x0,0} },
            { {(PD_REG *) PD_LC,0,0x0,1}, {0x0,0,0x0,0}, {0x0,0,0x0,0}, {0x0,0,0x0,0} },
            {0,0,0,0},
            {0,0,0,0},
            PD_ialu,
            0,0,0
        },
        {
            …
        }
```

The Simulation Loop (continued)

- All operations with the same schedule time are issued without updating the simulator cycle clock
- A MultiOp is a bundle of operations to be issued within the same cycle
  - VLIW or EPIC bundle
- A MultiOp separator is inserted by CODEGEN
  - PD_ACLOCK
  - Commits all actions pending in the action queue
  - Advances the simulation cycle count

The Simulation Loop (continued)

- Every simulated operation uses the PD_delay function to *queue* the pending *action* in an *action queue*
- An action is either a *write* action or a *call* action
  - Write actions copy a value from the simulator's local state to the processor state and is implemented as an assignment
  - Call actions transfers control to the specified function or block
- Note that actions that read values from the processor state for input operands are performed at issue time
  - i.e. Register and Memory reads

Branch Operations

- A call action may be the result of a *branch* or a *branch and link* operation
  - The source register is a pointer to a *block* or a *vector* associated with the target
- When branching to a block, the program counter is updated with the target block address
- When branching to a function, the vector transfers control to a new function
  - Function is either an *external* function or a *wrapper* function
    - external functions are those not processed by CODEGEN
      - Standard library function calls (i.e. printf)
    - wrapper functions are those processed by CODEGEN
      - Wrapper function will eventually transfer control back to the VMI

Speculative Execution Support

- To support speculative execution, a function for intercepting *exceptions* is defined
  - When an exception is raised, the (REBEL) id of the offending operation is recorded but the exception suppressed
- Exception is raised when the speculative bit on any of the source operands is set to true *and* the current operation is not issued speculatively
  - CODEGEN flags speculative operations

Speculative Execution Support (continued)

```
function void PD_simulate(Op opsTable[])
{
        for (Op *opPtr = opsTable[0]; !done; opPtr++)
        {
                if (!(opPtr->Mask | opPtr->Speculated)) {
                        /* if one of the sources was the result of a speculated instruction
                         * that caused an exception, raise the exception now
                         */
                        if (PD_has_speculated_sources(opPtr)) {
                                PD_raise_exception(opPtr->Rebel_id, 0);
                        }
                        else {
                                /* store the current operation's id in case of a delayed exception */
                                PD_DO_NOT_TRAP_RAISED_EXCEPTION = opPtr->Rebel_id + 1;
                        }
                }
                else {
                        PD_DO_NOT_TRAP_RAISED_EXCEPTION = 0;
                }

                /* perform the operation */
                *(opPtr → op)(opPtr);
        }
}
```

CODEGEN – The Simulation Front-End

CODEGEN Overview

- For each application source file specified a set of low-level *C* files are generated

xyz.c ⇒ xyz.el

xyz.inc	external, global, structure and union declarations as well as simulation library variables
xyz.tbls	one array of *Op* structures for each function declared in the source file **xyz.c**
xyz.c	this file includes the **xyz.inc** and **xyz.tbls** filesand also declares *wrapper* and *vector* functions

A **benchmark_data_init.simu.c** file is also generated for the entire application. It contains the definition of a function that initializes the global data for this application.

CODEGEN Overview

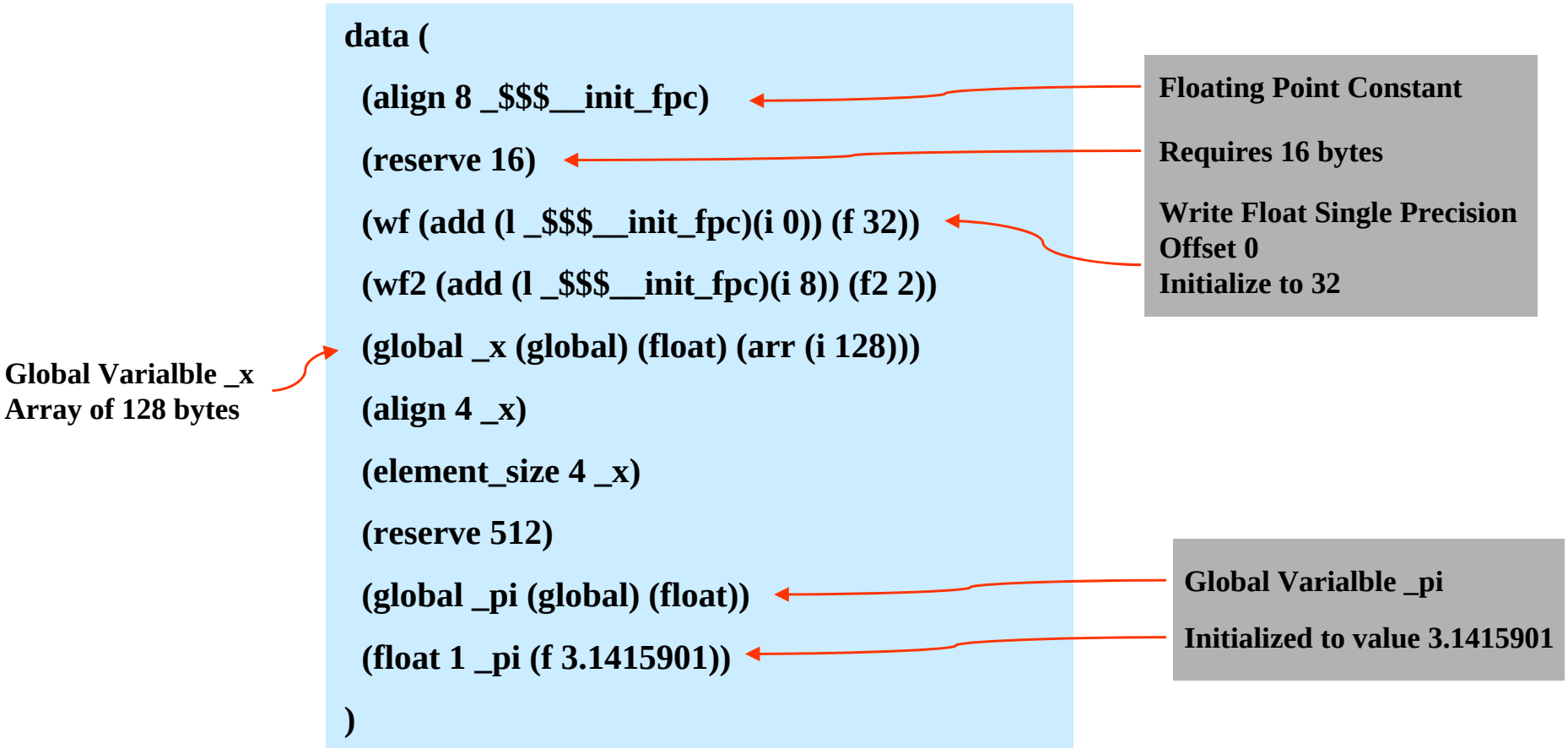
- A REBEL input file consists of multiple DATA and CODE sections
- Each DATA section is translated to corresponding *C* global data declarations
- Each operation in the CODE section is translated to an equivalent pseudo-operation
  - Branch operation require special translation
  - Instrumentation operations and MultiOp separators are inserted during the translation

CODEGEN Components

- DATA processor
  - Declares global variable and constants extracted from source file
- CODE processor
  - Translates a REBEL operation to an equivalent PD_OP
  - Inserts MultiOp separators and instrumentation code
- Vectorizer
  - Generates vector functions
- Packager
  - Generates wrapper functions
- Integrator/Emmitter
  - Generates the pseudo-executables

DATA Processor

- The REBEL data section contains variable names and layout information
  - Extracted from the host compiler preprocessor



DATA Processor (continued)

- simu_el_data_processor.cpp parses and translates the REBEL data sections to *C* declarations
  - Declares all global data variables and constants
  - Declares local floating point constants
  - Generates the *externs* list
  - Generates the function prototype list
  - Generates a function to initialize any variables if necessary
    - This function is invoked only once when the simulation begins

Code Processor

- The Code Processor translates the Rebel input to a C equivalent
  - Rebel operations are translated to a PD_OP data structure
- Instrumentation code is inserted
- Detects operation boundaries and inserts the ACLOCK operation

Operation Components

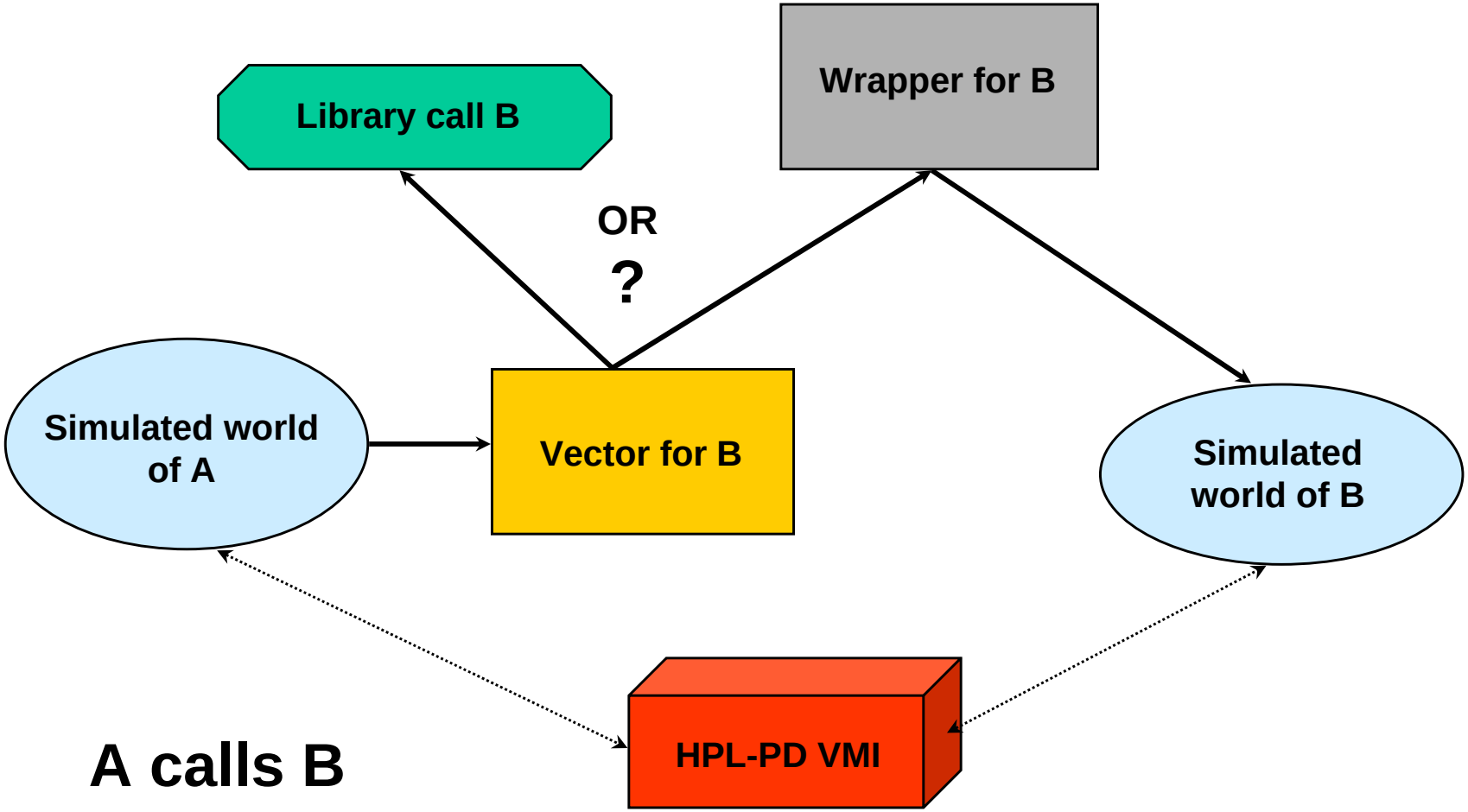
- A REBEL operation consists of
  - Operation id
    - Unique id for information tracking
  - Set of target operands
  - Set of source operands
  - Predicate operand

  Mapped to equivalent PD_PORTs
  - Schedule time
  - OPCODE
    - Information tracking and statistic gathering
  - Attributes (i.e. parameter layout)
  - Flags (i.e. spill operation, speculated operation)

Vector-Wrapper Mechanism

- Allows the simulated code to interact with native *C* code
  - A *vector* is the only exit interface from the HPL-PD environment
  - A *wrapper* is the only entry interface to the HPL-PD environment
- Calls to external and library functions are made completely transparent
  - Eliminates the need for a linker/loader for the HPL-PD instruction stream
  - Native compiler does the linking

Vector-Wrapper Illustration



Vectorizer

- For each function called from the REBEL input file a vector function is generated
  - Transformation code to convert from HPL-PD registers to native function parameters
  - Executes actual call to target function
  - Transformation code to convert the returned function values to HPL-PD registers

Vectorizer Example

C function to which a call is being made

```
int foo ( )
{
    .
    test (p1, p2);
    .
    .
}

int test (int b, double c)
{
    .....
}
```

Vectorized function

```
int  _vector_test (void)
{
   *PD_Int_Result =
              test (*PD_Int_Arg_1,
                    *PD_Dbl_Arg_1);

}
```

Packager

- For every function defined in the input REBEL file a corresponding wrapper is
  - Maps the incoming parameters to the HPL-PD registers and stack
  - Initializes the HPL-PD VMI if necessary
  - Invokes the HPL-PD VMI
  - Upon function return, the HPL-PD registers are mapped to the native parameters

Wrapper Example

The Original C function in source

```
int f( )
{
    .
    test(p1, p2, p3);
    .
    .
}

int test(int b, double c)
{
     ....
}
```

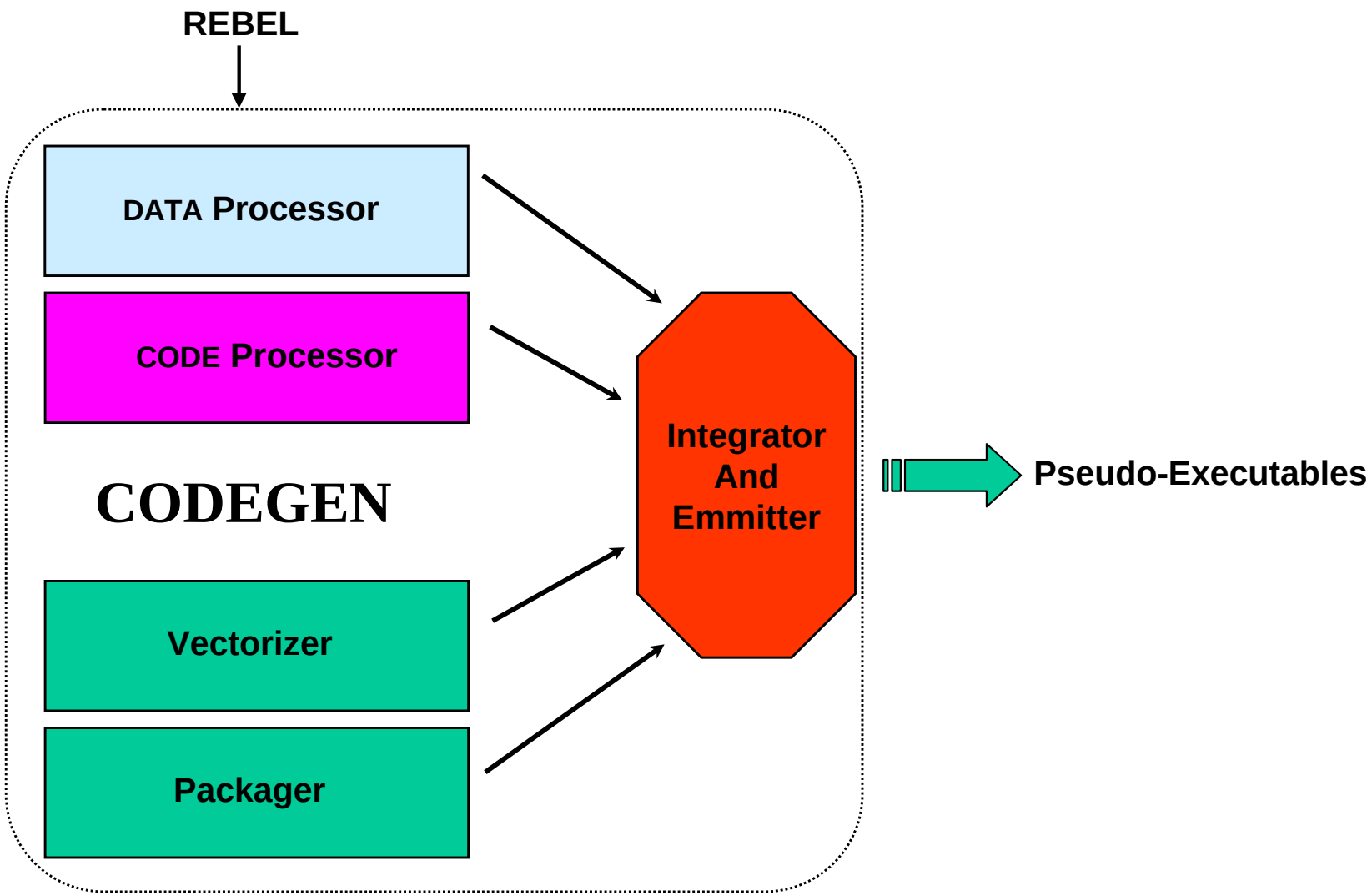
Generated wrapper

```
int test(int b, double c)
{
    *PDS_Int_Arg_1 = b;
    *PDS_Dbl_Arg_1 = c;

    PD_simulate(test_opsTable)

    return *PD_Int_Result;
}
```

Putting It All Together



Integrator/Emmitter

- Generates the xyz.c file
  - *#include*s xyz.inc and xyz.tbls
  - Contains the wrapper and vector functions
- Generates the xyz.inc file
  - Includes externs, global variables, floating point constants and global initialization function
- Generates the xyz.tbls file
  - Includes the OpsTables for the functions in xyz.c

Trimaran Extensions and Support

Reconfigurable Cache Simulator

- Smart Memory And Cache Hierarchy Simulator
  - SMACHS
- Memory hierarchy is specified using MDES
- SMACHS can be easily reconfigured and/or extended
  - User may *inherit* and customize cache objects
  - Default behaviors are easy to override

SMACHS Functionality Overview

- Accurate fine-grain cycle-by-cycle simulation
- Simulates DATA or INSTRUCTION cache
  - instruction cache simulation requires instruction format specification
- Parametric
  - uses HMDES to describe structure and behavior
  - user may choose to replace existing data management policies

Functionality Overview (continued)

- Fine-grain specification of a memory unit
  - Type of memory
    - Main memory or cache memory
  - Specify connectivity
    - Memory hierarchy layout, port and bus width
  - Direct mapped or N-way associative
  - Size specification
    - Capacity and line size
  - Transaction latencies
    - Includes port and bus latencies
  - Behavioral aspects
    - e.g. write-thru or write-back

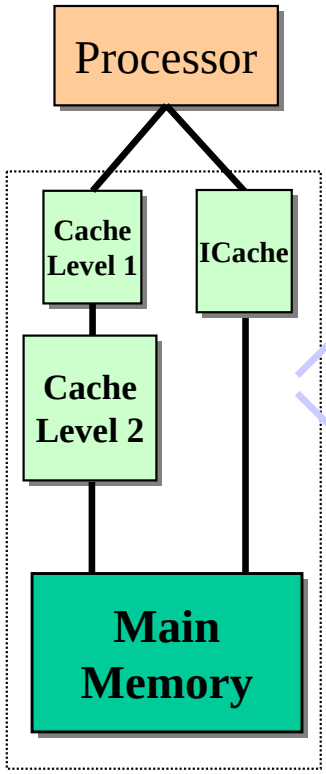
Functionality Overview (continued)

- Simulates complex cache interactions
  - Multiple cache levels, including multiple caches on the same level
- Multiple modes of operation
  - Trace-driven simulation
  - Direct interface to HPL-PD simulator
- Documentation
  - Tech-Report in progress

SMACHS - GUI Support

- Integrated into the Trimaran Laboratory
- Functionality includes
  - Memory hierarchy specification
    - Creating, editing and compiling
  - Trace selection
  - Off-line simulation
  - Statistical analysis
    - General statistics include total computation cycles, stall cycles, number of cache hits and misses
    - Operation specific statistics report the hit and miss ratios as well as the average latencies in delivery data for specific operations

Memory Hierarchy and HMDES

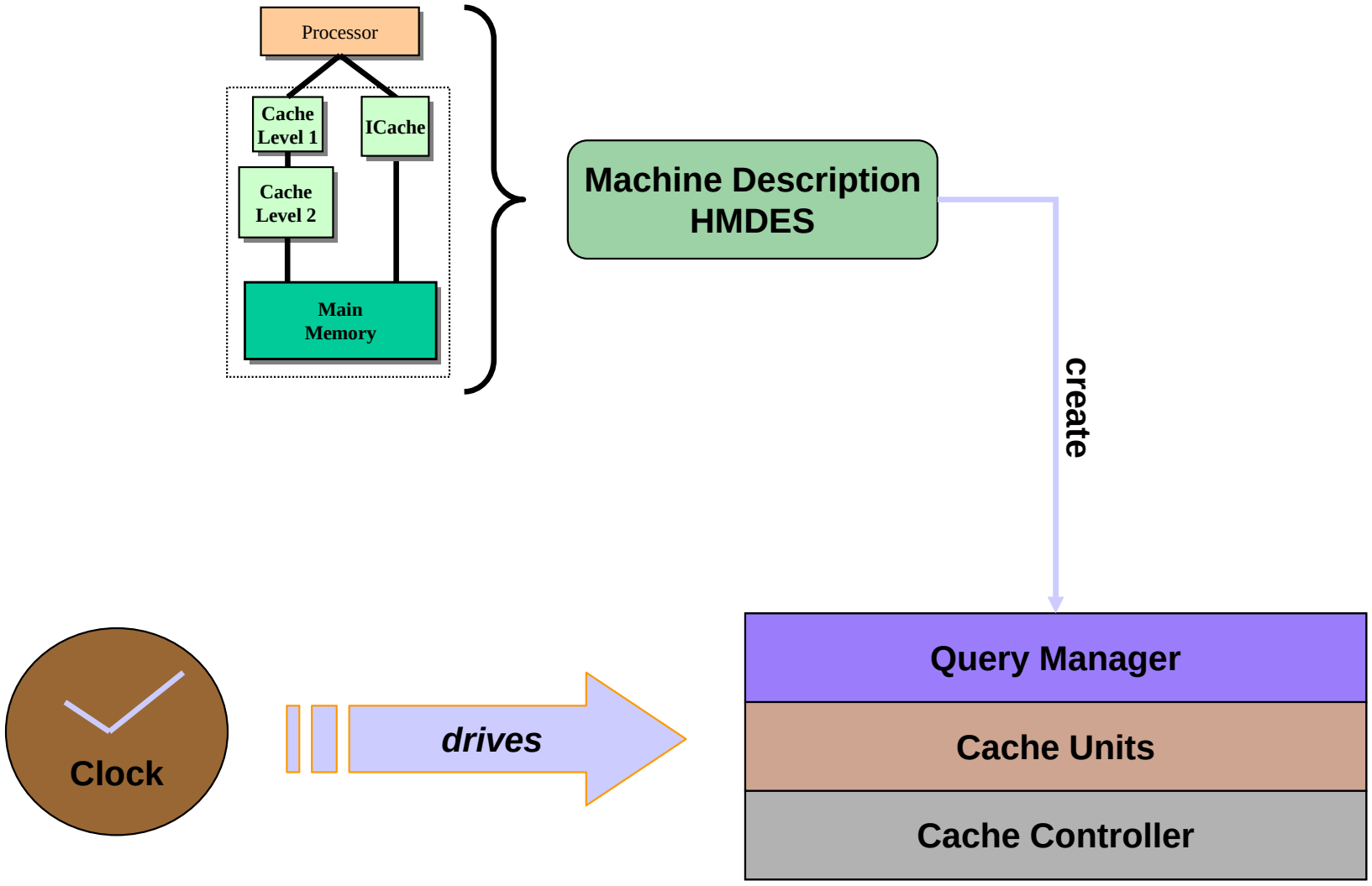


memory architecture

```
SECTION Cache
{
    …
    // L1Dcache
     L1D(type(Basic_Cache)
            num_lines(64)
            line_size(32)
            associativity(${assoc_direct_mapped})
            replacement(LRU)
            load_hit_latency(1)
            load_miss_latency(1)
            store_hit_latency(1)
            store_miss_latency(1)
            turnaround_latency(0)
            max_misses(1));
    …
}
```

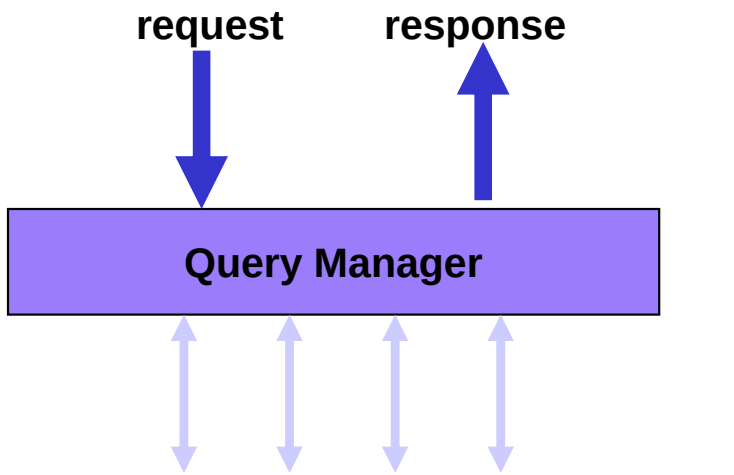
```
SECTION Port
{
    L2toMainPort(width(32) latency(5) up(L2) down(MAIN));
    L1DtoL2Port(width(16) latency(2) up(L1D) down(L2));
    L1ItoMainPort(width(1) latency(1) up(L1I) down(L2));
    MMUtoL1DPort(width(1) latency(0) up(MMU) down(L1D));
    MMUtoL1IPort(width(1) latency(1) up(MMU) down(L1I));
}
```

SMACHS Design Overview



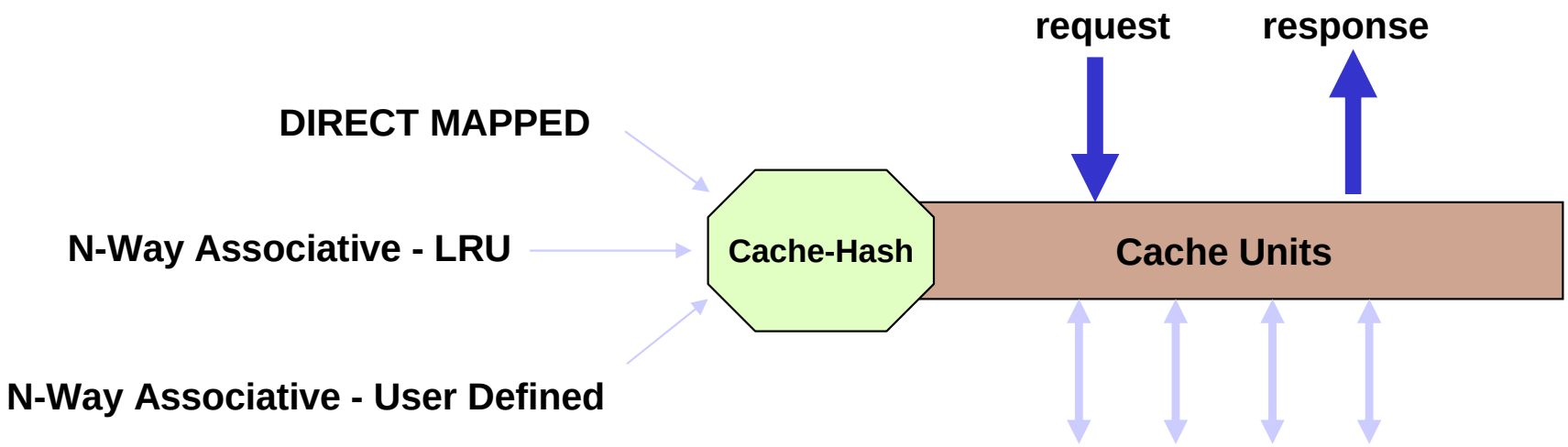
Query Manager - QM

- Interface for all memory requests
- Simulator or Trace driven
- Multiple output ports



Cache Units

- Parametric cache units
  - Line size, number of lines, associativity, latencies, replacement policy, etc…
  - Cache-Hash attribute
  - Multiple input/output ports



Cache Controller

- Specifies complex behavior
  - Next-Hop-Table as an example
    - Central request routing table
    - Can be used to dynamically dispatch a request along different routes