Harcom

Hardware complexity model for microarchitecture exploration

Pierre Michaud

June 25, 2025

Contents

Chapter 1

Introduction

Microarchitecture exploration is generally conducted with performance simulators written in general-purpose programming languages such as C or C++. For example, gem5 [5, 3] and ChampSim [6, 2] are two popular open-source performance simulators. A performance simulation outputs various statistics, such as execution time, number of cache misses, number of branch mispredictions, etc. A performance simulator does not need to simulate all the details of the hardware implementation. It is often sufficient to simulate the events that can impact performance significantly, such as cache misses, branch mispredictions, data dependences, etc. Performance simulators often use approximations and abstractions. This is what allows them to simulate the execution of many instructions in a short amount of time, which is important for estimating millisecond-scale performance and for design space exploration.

People using performance simulators are generally engineers, researchers or students, hereafter referred to collectively as *microarchitects*. In a typical situation, a microarchitect needs to study the effects of modifying a part of the microarchitecture. Performance simulators are easily modifiable to conduct such study. The constraints for modifying the simulator are generally few besides those of the programming language itself (e.g., C++). Microarchitects generally try to achieve their goal with minimal modifications to the simulator, so they are practically constrained by how the simulator is structured and how the part they want to modify communicates with the rest of the simulator. Otherwise, microarchitects can use whatever approximation or abstraction they like. Such flexibility comes with a drawback: there is no guarantee that a modification corresponds to realistic hardware.

In general, microarchitects are aware of hardware constraints and try to simulate realistic mechanisms. Nevertheless, assessing the hardware complexity of a mechanism which only exists as a piece of C++ code in a performance simulator can be difficult. Hardware complexity is a multidimensional quantity including silicon area, energy consumption and delay. A simple, oft-used estimate of hardware complexity is the amount of storage (typically, SRAM capacity) used by a mechanism. Indeed, the silicon area, energy and access latency of an SRAM increases with its size, and a substantial part of the hardware complexity of processors comes from on-chip SRAMs. Still, there is more to hardware complexity than storage. For instance, the delay of a branch predictor depends not only on the size of its SRAMs but also on the logic circuits processing the information retrieved from the SRAMs.

Microarchitects, especially in academia, often use high-level complexity models such as CACTI [10, 1] and McPAT [7, 4]. These tools are distinct from the performance simulator: the microarchitect must manually configure CACTI/McPAT to reflect the hardware modification. Moreover, these tools have limited configurability. For instance, the branch predictor modeled in McPAT is the one implemented in the Alpha 21264. Modeling a different predictor requires

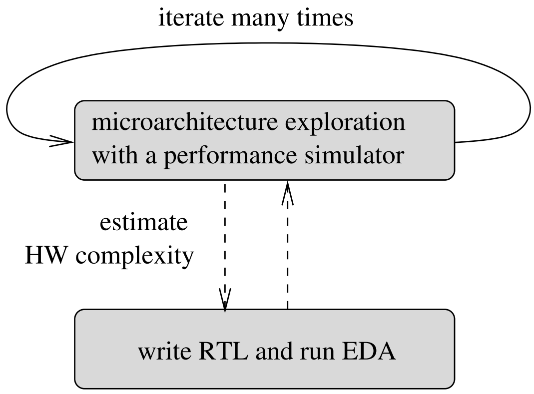

Figure 1.1: Hardware complexity estimation is off the main microarchitecture exploration loop.

to hack McPAT's source code.

The most general solution for estimating the hardware complexity of a microarchitectural part is to use a hardware description language (HDL) such as SystemVerilog, write a RTL (Register Transfer Level) description of the part and run EDA (Electronic Design Automation) tools to assess the hardware complexity. However, this is a time-consuming process, and hardware complexity estimation is generally off the main microarchitecture exploration loop (Figure 1.1).

Harcom is not a HDL. The goal is not to synthesize hardware. The purpose of Harcom is to provide a hardware complexity model directly inside the performance simulator. The hope is that Harcom improves the process of selecting solutions to implement in HDL and reduces the burden of designers.

Harcom tries to find a useful middle ground between several contradictory objectives: hardware complexity model accuracy, simulation speed, flexibility and ease of use. This implies tradeoffs that make Harcom's complexity model a very rough approximation of what a designer can obtain with RTL/EDA. Nevertheless, an approximate model can still be useful if it provides sufficient qualitative accuracy and if the microarchitect understands the sources of error and the model's limitations.

Chapter 2

Overview of Harcom

Harcom is a C++20 library consisting of a single header file ("harcom.hpp"). Most performance simulators today are written in C++, so incorporating Harcom in existing simulators should be straightforward.

Harcom's basic data type is called `val`. A `val` object is declared with a parameter N and represents an N-bit integer value[1] which can also be viewed merely as a bundle of N bits. Listing 2.1 shows a simple C++ program using Harcom's vals. Each val has a value and a timing in picoseconds which are both printed with the method `print()`. Vals *x* and *y* both have a null timing, as they are initialized from hardwired values, i.e., values known when designing the hardware. However, operations on vals generally increase the timing: val *z*, the sum of *x* and *y*, has a timing corresponding to the latency of an 8-bit adder. In the general case, the timing of the result of a two-input operation is the maximum of the timing of the two inputs plus the latency of the hardware operator, as illustrated in Figure 2.1. The function `panel.print()` prints the total number of transistors used and the total energy consumption.

Figure 2.2 illustrates a typical usage of Harcom, where only the part of the performance simulator modeling the processor component that we want to study is rewritten to use Harcom vals in place of C++ integers. The rest of the simulator remains unchanged. The outputs of the component are vals, whose timing, along with the total number of transistors and total energy consumption, is a measure of the hardware complexity of the component.

Performance simulators sometimes use abstractions that do not correspond to an actual hardware implementation. In order to estimate hardware complexity, Harcom restricts what users can do with vals. These constraints can be called the *Harcom language*.

In particular, the actual value of a `val` is a private member of the `val` C++ class: trying to read or write this value directly triggers a compilation error. While C++ makes it possible to circumvent the *private* access specifier if that is the user's intention, this is, hopefully, unlikely

[1] Future versions of Harcom might provide floating-point values.

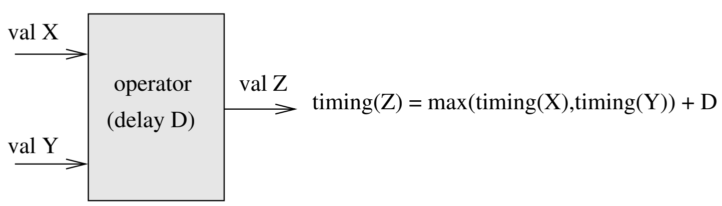

Figure 2.1: The timing of the result of a two-input operation is the maximum of the timing of the two inputs plus the latency of the hardware operator.

Listing 2.1: A simple C++ program using Harcom's vals

```
#include "harcom.hpp"
using namespace hcm; // Harcom namespace

int main()
{
  val<8> x = 1; // 8-bit unsigned integer
  val<4> y = 2; // 4-bit unsigned integer
  auto z = x + y; // 9-bit unsigned integer
  z.print("sum=");
  panel.print();
}

// prints on the standard output:
//   sum=3 (t=42 ps)
//   storage (bits): 0
//   transistors: 406
//   dynamic energy (fJ): 9.04
//   static power (mW): 0.000152
```

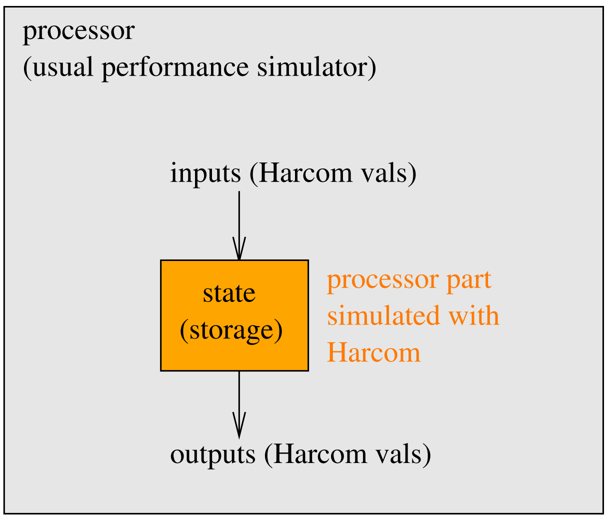

Figure 2.2: Harcom's typical usage: only the part of the performance simulator modeling the processor component that we want to study is rewritten.

to happen accidentally.

Nevertheless, the outputs of a component modeled with Harcom must be communicated to the rest of the performance simulator as normal C++ integers. Harcom distinguishes the *user* from the *superuser*. The superuser is whoever owns (i.e., can modify) the class called `harcom_superuser`. While the user is constrained by the Harcom language, the superuser can access private members and is responsible for implementing the interface between the component modeled with Harcom and the rest of the simulator. For example, in the context of a branch prediction championship, the superuser would be the championship's organizers and the user would be a contestant. Otherwise, the user and superuser might be a single person or a group of people willing to use Harcom the way it was intended to be used, as explained in this document.

Chapter 3

The Harcom language

The Harcom language is not a proper programming language, it is just C++ programming with Harcom vals. However, there are strong constraints associated with vals, and programming with them can be viewed as a distinct language. A general rule of the Harcom language, call **no-hidden-cost**, is that the hardware cost of each statement is evaluated.

Throughout this document, *rightmost* bits refers to the least significant bits of an integer and *leftmost* bits refers to the most significant bits.

3.1 Harcom data types

3.1.1 The `val` type

The `val` type represents a **transient** value, i.e., a value existing at a certain time, and with a limited lifetime. A `val` takes two template parameters N and T, where N is the number of bits and T is the underlying C++ **integer** type. For example:

```
val<10,u64> x = 1; // 10 bits; underlying type is std::uint64_t;
val<6,i64> y = -1; // 6 bits; underlying type is std::int64_t;
val<8> z = 1; // equivalent to val<8,u64>
```

Note that `u64` and `i64` are convenient aliases for `std::uint64_t` and `std::int64_t` that we use throughout this document.[1] While it is possible to use smaller integer types (`int`, `short`,...) to save a little memory, `u64` and `i64` are sufficient most of the time. If type T is omitted in the declaration, the underlying type is `u64` by default (see the example above). The value of N must not exceed type T's number of bits. In any case, N must not exceed 64.

A `val` must be initialized with a value, which can be a C++ integer literal, a C++ integer variable, another `val` or a `reg` (see section 3.1.2). When initializing from another `val` (or `reg`), the destination and source vals do not need to have the same size:[2] the value is truncated if the source val is longer than the destination val, it is sign extended if shorter:

```
val<8> x = 0b11111111; // 255
val<4> y = x; // truncated: 0b1111 (15)
val<8> z = y; // sign extended: 0b00001111 (15)
```

The Harcom user cannot change the value of a val.[3]

[1] They are actually defined in the "harcom.hpp" header file.

[2] They do not need to have the same type either: Harcom uses the same implicit conversions as C++.

[3] Attempting to change the value of a val triggers a compilation error.

name	type	description	example
`size`	C++ int	number of bits	
`maxval`	C++ int	maximum value	`val<4> x = val<4>::maxval`
`minval`	C++ int	minimum value	`val<4,i64> x = val<4,i64>::minval;`
`print`	function	print value	see Section 3.1.1
`printb`	function	binary printing	see Section 3.1.1
`fanout`	function	set fanout	`val<1> x = 1;` `x.fanout(hard<4>{});`
`fo1`	function	set fanout of 1	`val<3> x = 1;` `val<3> y = x.fo1() + 1;`
`make_array`	function	make an array from the value bits	`val<12> x = 0b101011110011;` `auto A = x.make_array(val<4>{});`
`reverse`	function	reverse bits	`val<8>{43}.reverse().printb();`
`rotate_left`	function	rotate bits	`val<8>{43}.rotate_left(-1).printb();`
`ones`	function	bit count	`val<8>{43}.ones().print();`
`one_hot`	function	reset all bits but the rightmost 1	`val<8>{44}.one_hot().printb();`
`replicate`	function	replicate the value (generate an array)	`val<1> x = 1;` `x.replicate(hard<4>{}).print();`

Table 3.1: Public members of class `val` besides constructors. The functions highlighted in red have a non-null hardware cost. Class `reg` inherits the same members.

While the value of a val is a private member that the Harcom user should not try to access directly, it is possible to print the value to the standard output, in decimal or binary representation:

```
x.print(); // prints "255 (t=3 ps)"
y.printb(); // prints "1111 (t=6 ps)"
```

Functions `print` and `printb` have default parameters that can be overridden:

```
z.print("z=","\n",false,std::cerr);
// prints "z=15" to the error stream
```

Table 3.1 lists the public members of class `val` that the Harcom user can access (constructors are omitted). Functions with a non-null hardware cost are highlighted in red.

3.1.2 The `reg` type

The `reg` type is derived (in the C++ sense) from the `val` type. A `reg` (for *register*) represents a **persistent** value, i.e., a value that is associated with storage. Unlike a val, a reg can be modified:

```
reg<4> x = -1; // 4-bit unsigned register, initialized with 15
reg<4,i64> y = x; // 4-bit signed register, initialized with -1
x = 0; // a reg can be modified
```

If a reg is not initialized explicitly, it is initialized implicitly with zero. Regs must obey the following two rules:

- **All regs must have the same lifetime**. That is, a reg cannot be created after another reg has been destroyed.[4]

- **A reg can be modified at most once per clock cycle**.

Violating these rules triggers an error at execution time. Besides the properties mentioned above, a `reg` is akin to a `val`, as illustrated by the example below:

```
auto increment = [](val<2> &x) -> val<2>
{
  return x+1;
};
reg<2> y = 1;
y = increment(y); // equivalent to y=y+1
y.print();
```

In this document, the term **valtype** refers to both vals and regs.[5] The public members of class `val`, listed in Table 3.1, are also public members of class `reg`.

3.1.3 The `arr` type

The `arr` type represents an array of valtype objects. An `arr` takes two template parameters T and N, where T is a valtype and N is an unsigned integer:

```
arr<val<3>,4> A = {1,2,3,4};
A[2].print(); // print the third element
arr<val<3>,4> B = [](u64 i){return i+1;};
arr<reg<1>,4> C = B;
C.print(); // print all the elements
```

The subscript operator `[]` returns a reference to a particular element (second line). The first element has index 0 (like C arrays).

In the example above, array B is initialized from a C++ lambda (third line). It is sometimes necessary to use a lambda or a function to initialize an array of vals, as vals, unlike regs, cannot be changed after their creation.

Table 3.2 lists the public members of class `arr` that the Harcom user can access (constructors and operators are omitted). Functions with a non-null hardware cost are highlighted in red. The functions `concat`, `make_array`, `shift_left`, `shift_right` treat array elements as consecutive chunks of a bit vector. The first array element (index 0) corresponds to the rightmost bits of this bit vector.

The array assignment operator is public. However, if the Harcom user tries to modify an array of vals, this triggers a compilation error. The subscript operator `[]` already mentioned in Section 3.1.3 takes C++ integers as argument. An array with a single element is implicitly convertible to a `val`:

```
arr<val<4>,1> A = {10};
val<4> x = A;
```

[4]To make sure that this rule is not violated, it is sufficient (but not necessary) to declare all regs as static variables.

[5]The "harcom.hpp" header file defines a C++ concept of that name (`static_assert(valtype<reg<8>>);`)

name	type	description	example
size	C++ int	number of elements	
print	function	print all the elements	same syntax as valtype
printb	function	binary printing	same syntax as valtype
select	function	read a selected element	arr<val<2>,4> A = {1,3,0,2}; A.select(A[1]).print();
concat	function	concatenate all bits into single val	arr<val<3>,3> A = {0b000,0b111,0b010}; A.concat().printb();
fanout	function	set fanout	A.fanout(hard<16>{});
fo1	function	set fanout of 1	A.fo1().concat().printb();
append	function	generate array with one extra element	A.append(7).print();
truncate	function	truncate the array	A.truncate(hard<2>{}).print();
make_array	function	concatenate all bits & make new array	arr<val<3>,2> A = {0b000,0b111}; A.make_array(val<2>{}).printb();
shift_left	function	insert bits, shift left	arr<val<3>,2> A = {0b000,0b111}; A.shift_left(val<2>{0b11}).printb();
shift_right	function	insert bits, shift right	arr<val<3>,2> A = {0b000,0b111}; A.shift_right(val<2>{0}).printb();
fold_xor	function	XOR all elements	arr<val<3>,3> A = {0b100,0b110,0b111}; val<3> x = A.fold_xor();
fold_or	function	OR all elements	val<3> x = A.fold_or();
fold_and	function	AND all elements	val<3> x = A.fold_and();
fold_xnor	function	XOR all elements, then complement	val<3> x = A.fold_xnor();
fold_nor	function	OR all elements, then complement	val<3> x = A.fold_nor();
fold_nand	function	AND all elements, then complement	val<3> x = A.fold_nand();
fold_add	function	add all elements	arr<val<3>,3> A = {4,6,7}; val<5> x = A.fold_add();

Table 3.2: Public members of class arr besides constructors and operators. The functions highlighted in red have a non-null hardware cost. The functions concat, make_array, shift_left, shift_right treat array elements as chunks of a bit vector. The first array element (index 0) corresponds to the rightmost bits of this bit vector.

3.1.4 The hard type

The hard type represents hardware parameters, that is, values that are fixed and known at design time. It takes a single template parameter N which is the value of the hardware parameter. That is, object hard<N>{} represents value N. For example:

```
val<8> x = -1;
val<8> y = x << hard<4>{}; // shift left by 4 bits
```

In many situations, it is possible to substitute a C++ integer (variable or literal) for a hard parameter:

```
val<8> y = x << 4; // equivalent to y = x << hard<4>{}
```

While convenient, this is not always possible though. For example the modulo operation requires the modulus to be a hard parameter:[6]

```
val<4> x = -1;
auto y = x % hard<4>{};
```

While the use of C++ integers is allowed by the Harcom language, **the use of non-constant integers whose lifetime spans multiple clock cycles violates the no-hidden-cost rule**. However, the Harcom library does not enforce the no-hidden-cost rule (unfortunately). Compliance with the no-hidden-cost rule rests on the user's discipline. If an algorithm whose hardware complexity we seek to evaluate requires a modifiable persistent value, we must use a reg instead of a C++ integer.

3.2 The ram type

The ram type emulates a random access memory (RAM). It takes two template parameters T and N, where T is the type of data stored in the RAM and N is the memory size in number of such data. Type T can be val or array of val.[7] For example:

```
ram<val<3>,32> mem; // 3-bit data, 32 data
val<5> addr = 10;
val<3> data = 7;
mem.write(addr,data); // RAM write
val<3> readval = mem.read(addr); // RAM read
readval.print(); // prints 7
```

In the Harcom language, the value produced by an an operation on vals generally does not depend on the timing of the input vals. That is, the timing of inputs only affects the timing of the output, not the value. However, there is one exception, which is when reading a RAM. Harcom's RAM model assumes that the time at which a write occurs is the maximum of the address and data timings. When reading a RAM at a given address A, the data returned by the read operation is the data written by the most recent write whose timing is less than or equal to the timing of A. In other words, we cannot read a value that will be written in the future. For example:

```
ram<arr<val<64>,2>,1024> mem;
val<10> addr = 100;
arr<val<64>,2> data = {addr,addr+1};
```

[6] A compilation error occurs if the modulus is a C++ integer.
[7] T is the type of the data returned by a read operation.

name	description	example
`write`	`write(addr,data)` writes data (valtype or arr) at address addr	`ram<arr<val<64>,2>,256> mem;` `val<8> addr = 100;` `arr<reg<64>,2> data = {1,2};` `mem.write(addr,data);`
`read`	`read(addr)` returns the data stored at address addr	`data = mem.read(addr+1);`
reset	reset the RAM with zeros	`mem.reset();`
`print`	prints delay and energy	`mem.print();`

Table 3.3: Public functions of `class ram`.

```
mem.write(addr,data);
mem.read(addr).print(); // prints 0 0
```

The RAM write is effective when the addition operation (`addr+1`) is finished, which happens in the future compared to the RAM read operation. So the RAM read returns the old data, which is zero in this example (the value with which the RAM is automatically initialized).

RAMs must obey the following two rules:

- **All RAMs must have the same lifetime as regs**. That is, a RAM cannot be created after a reg or another RAM has been destroyed.[8]
- **Only a single RAM read and a single RAM write are allowed per clock cycle** (otherwise there is an error at execution).

Public members of class `ram` are listed in Table 3.3.

3.3 The `rom` type

The `rom` type emulates a read-only memory (ROM). It takes two template parameters T and N, where T is a `val` type (the type returned by a ROM read) and N is the ROM size in number of such vals. A rom object must be initialized at creation:

```
rom<val<3>,16> bitcount = {0,1,1,2,1,2,2,3,1,2,2,3,2,3,3,4};
val<4> bitvec = 7;
bitcount(bitvec).print(); // prints 3
```

The first element has index 0. The ROM is read with operator `()`. Despite the name, a ROM is not a memory but is akin to a function.

ROMs are initialized like arrays. In particular, they can be initialized from a function or a lambda:

```
rom<val<3>,16> bitcount = [](u64 i){return std::popcount(i);};
```

3.4 Arithmetic and logical operators

Many operators of the C language can be used with valtypes and have the same meaning as in C. These operators are listed in Table 3.4. Each operator takes one or two valtypes (`val`

[8]In other words, all storage (reg or RAM) must have the same lifetime.

<table>
<tr><th>operator</th><th>operation</th><th>input type</th><th>output type</th></tr>
<tr><td>==</td><td>equal</td><td rowspan="6">two vals of same size
or one val and one hard</td><td rowspan="6">val<1></td></tr>
<tr><td>!=</td><td>not equal</td></tr>
<tr><td>></td><td>greater than</td></tr>
<tr><td><</td><td>less than</td></tr>
<tr><td>>=</td><td>greater than or equal</td></tr>
<tr><td><=</td><td>less than or equal</td></tr>
<tr><td>&</td><td>bitwise AND</td><td rowspan="3">two vals or
one val and one hard</td><td rowspan="3">same as longest of the input vals</td></tr>
<tr><td>|</td><td>bitwise OR</td></tr>
<tr><td>^</td><td>bitwise XOR</td></tr>
<tr><td>~</td><td>bitwise NOT</td><td>one val</td><td>same as input</td></tr>
<tr><td><<</td><td>shift left</td><td rowspan="2">one val and
one hard shift count</td><td rowspan="2">same as input val</td></tr>
<tr><td>>></td><td>shift right</td></tr>
<tr><td>+</td><td>add</td><td rowspan="2">two vals or
one val and one hard</td><td rowspan="2">one bit longer than the longest input val</td></tr>
<tr><td>-</td><td>subtract</td></tr>
<tr><td>-</td><td>change sign</td><td>one val</td><td>same as input</td></tr>
<tr><td>*</td><td>multiplication</td><td>two vals or
one val and one hard</td><td>val with enough bits
(≤ 64 bits)</td></tr>
<tr><td>/</td><td>integer division</td><td rowspan="2">unsigned val dividend
hard divisor</td><td rowspan="2">val with enough bits</td></tr>
<tr><td>%</td><td>modulo (remainder)</td></tr>
</table>

Table 3.4: Arithmetic/logical operators. Inputs are valtypes or hard values. Outputs are vals. All operators have a hardware cost except <<, **unsigned** >>, and & and | with a **hard** value.

or reg) as input. Some binary operators allow to substitute a single hard value for an input valtype. Some binary operators *require* one of the two inputs to be a hard value. The ouput of an operator is always a val.

3.5 Free functions

Table 3.5 lists the free functions that are part of the Harcom language. The functions at the bottom of Table 3.5 are called "utilities" because they are written in the Harcom language. Harcom users could write them themselves, without superuser privilege. Their implementation is located at the end of the "harcom.hpp" file.

The execute_if function is an essential primitive allowing conditional execution. It takes two inputs: a valtype *mask* and a C++ lambda F that can have a C++ integer parameter i. The execute_if primitive executes F(i) for each i corresponding to a mask bit that is set. If F returns a val, execute_if returns an array of vals whose elements corresponding to null mask bits are zeros. For example, execute_if can be used to access a RAM conditionally:

```
ram<val<2>,64> mem;
val<1> cond = false;
val<6> addr = 42;
val<2> data = 3;
execute_if(cond,[&](){mem.write(addr,data);});
val<2> x = execute_if(cond,[&](){return mem.read(addr);});
x.print();
```

When the mask bit is null, no energy is consumed[9] and the storage (regs/RAMs) written by F

[9]The transistor count is incremented though.

name	description	example
`a_plus_bc`	compute $a+b\times c$	`a_plus_bc(a,b,c).print();`

`concat`	concatenate multiple vals into a single val	`val<3> left = 0b111;`
		`val<4> right = 0b0011;`
		`val<7> z = concat(left,right);`

`select`	`select(cond,x1,x0)` equals `x1` if `cond` is true, `x0` otherwise	`val<1> incr = true;`
		`val<4> x = 0;`
		`val<4> y = select(incr,val<4>{x+1},x);`

`execute_if`	`execute_if(mask,F)` executes the C++ lambda F for each mask bit that is set	`val<4> x = 11;`
		`auto pp = execute_if(x,`
		`  [&](u64 i){return val<8>{x}<<i;});`
		`pp.fold_add().print("x^2=");`

utilities		
`absolute_value`	if signed value is negative, make it positive	`val<8,int> x = -3;` `absolute_value(x).print();`

`encode`	encode a one-hot bit vector	`val<8> ask = 0b01000100;`
		`val<8> onehot = ask.one_hot();`
		`val<3> index = encode(onehot);`

`fold`	`fold(A,op)` folds array `A` with binary associative operation `op`	`auto max = [] (val<4> x, val<4> y) {`
		`  return select(x>y,x,y);`
		`};`
		`arr<val<4>,4> A = {8,2,13,7};`
		`fold(A,max).print();`

`scan`	`scan(A,op)` yields the prefix-sum array of array `A` with binary associative operation `op`	`auto add = [] (val<4> x, val<4> y) {`
		`  return x+y;`
		`};`
		`arr<val<4>,8> A = [](){return 1;};`
		`scan(A,add).print();`

Table 3.5: Free functions. They all have a hardware cost except `concat`. For `execute_if`, `fold` and `scan`, the hardware cost depends on the function/lambda that is passed as argument.

is actually unmodified. However, every attempt to write a reg or read/write a RAM, **even when the mask bit is null**, is subject to the one reg write and one RAM read/write per cycle limit. Consequently, writing the same reg or accessing the same RAM inside F is not possible unless the mask is a single bit.

3.6 The hardware cost of reading values

The Harcom user focuses first on the functional behavior of the microarchitectural algorithm, which is generally independent of timing[10] unless the timing information is used explicitly by the algorithm. Once the algorithm is bug-free and works as expected, the Harcom user tries to reduce the hardware cost.

Reading a `val` or a `reg` is associated with a hardware cost, especially a read delay. Harcom does not know at compile time how many times a named value will be read. Therefore, a pessimistic situation is assumed where each read incurs an extra delay, which Harcom models as that of a fanout-of-two (FO2) inverter. The read delay increases linearly with the number

[10] Except for RAM reads, as explained in Section 3.2

of reads.[11] While the delay of a single read can be considered negligible, the accumulation of read delays can be quite significant.

In real circuits, a high fanout (i.e., reading the same value many times) means that we must drive a high capacitance, which takes some time. However, with optimal buffering and gate sizing, delay grows roughly logarithmically with fanout [8, 9], not linearly.

3.6.1 The `fanout` function

Reading an unnamed value (aka *rvalue*) incurs no hardware cost, as it is known at compile time that such value will be read only once. However, it is not known at compile time how many times a named value (aka *lvalue*) is read. To make the delay of reading a named value logarithmic instead of linear, the Harcom user must use the `fanout` function:

```
val<4> x = 1;
x.fanout(hard<8>{}); // make delay logarithmic
arr<val<1>,8> A = x.replicate(hard<8>{});
A.print();
```

If the value is actually read more than what was promised with the `fanout` function, no error is triggered. Instead, the read delay simply grows linearly after the initial logarithmic growth. Compiling with the option `-DCHECK_FANOUT` forces an error at execution if the actual fanout exceeds the declared one.

3.6.2 The `fo1` function

Whenever possible,[12] transient values (vals) that are read only once should remain unnamed. Nevertheless, for program readability, the Harcom user may wish to give a name to a `val` even though it is read only once. In this situation, if the read delay is deemed non-negligible, it is possible to use function `fo1` to "unname" a named value:

```
val<4> x = 1;
arr<val<1>,8> A = x.fo1().replicate(hard<8>{});
A.print();
x.print(); // x has been reset!
```

Attempting to apply `fo1` to a `reg` triggers a compilation error (a `reg` cannot be unnamed).

The `fo1` function should be used very cautiously. By using `fo1`, the programmer promises that the value will not be read again. To make it impossible to obtain an unrealistic advantage from a misuse of `fo1`, a read through `fo1` is destructive, that is, the value is reset.

The compiler option `-DFREE_FANOUT` disables destructive reads and removes all read delays. This option is useful for detecting some misuses of `fo1` and for checking whether there is much to gain from optimizing fanouts.

3.6.3 The `split` type

The `split` type allows to split the bits of a `val` into two parts without any read penalty:

```
val<8> x = 0b11000100;
split<3,5> y = x.fo1();
```

[11] This corresponds to chaining FO2 inverters.

[12] Function parameters must have a name, even if they are read only once.

```
y.left.printb(); // 3 bits (0b110)
y.right.printb(); // 5 bits (0b00100)
```

Or, using structured binding (C++17):

```
auto [left,right] = split<3,5>(x.fo1());
left.printb();
right.printb();
```

Chapter 4

Using Harcom

Figure 4.1 gives a contrived example of utilization of Harcom. In this example, the function `collatz`, written in the Harcom language, is the function whose hardware complexity we seek to evaluate. Notice that variable `value` is a C++ integer whose lifetime spans multiple clock cycles. The hardware cost of this modifiable persistent value is not modeled, on purpose. As `value` is not used directly in function `collatz`, the no-hidden-cost rule is not violated (Section 3.1.4).

4.1 The panel

The *panel* is a global object. The panel contains some global variables that the user can read. For example, variable `energy_fJ` gives the total energy (in femtojoules) dissipated so far:

```
val<64> x = 0;
x+1;
panel.energy_fJ.print("val+hard:␣");
f64 e = panel.energy_fJ;
x+x;
std::cout << "val+val:␣" << panel.energy_fJ - e << std::endl;
```

Global variables can be read but cannot be modified by the Harcom user. Only the superuser can modify them. Global variable are implicitly convertible to their C++ underlying type, which is `f64` (i.e., `double`) for `energy_fJ` and `u64` for all the other variables. Table 4.1 lists the panel variables and functions that the user and the superuser can utilize.

4.2 The superuser

The superuser is whoever can modify the class `harcom_superuser`. Class `harcom_superuser` is defined in the global namespace.

The superuser can transform Harcom data into C++ integers, which is necessary to implement the interface between the part of the simulator implemented in the Harcom language and the rest of the simulator. The superuser can also write or read the timing associated with a Harcom data. Table 4.2 lists the private functions of classes `val`/`reg` and `arr` that the superuser can use.

The timing of a Harcom data can be set by the superuser with the `set_time` function. The timing of a `val` can also be set at construction time if the initialization is from a C++ integer:

```
#include "harcom.hpp"

struct harcom_superuser {
  uint64_t value = 27;

  harcom_superuser() {
    hcm::panel.clock_cycle_ps = 200;
  }

  hcm::val<64> collatz(hcm::val<64> n) {
    // function whose hardware complexity we seek to evaluate
    using namespace hcm;
    constexpr auto two = hard<2>{};
    constexpr auto three = hard<3>{};
    val<1> odd = n % two;
    val<64> inc = execute_if(odd,[&](){return three*n+1;});
    val<64> dec = execute_if(~odd,[&]()->val<64>{return n/two;});
    return select(odd,inc,dec);
  }

  void one_cycle() {
    auto [v,t] = collatz(value).get_vt();
    if (t >= hcm::panel.clock_cycle_ps)
      std::cerr << "timing␣failure:␣" << t << "\n";
    assert(t < hcm::panel.clock_cycle_ps);
    hcm::panel.next_cycle();
    value = v;
  }
} hsu;


int main()
{
  while (hsu.value != 1)
    hsu.one_cycle();
  hcm::panel.cycle.print("total␣cycles:␣");
  hcm::panel.print();
}
```

Figure 4.1: A contrived example of utilization of Harcom.

name	type	description	example
`clock_cycle_ps`	variable	clock cycle (picoseconds)	`panel.clock_cycle_ps.print();`
`cycle`	variable	current cycle	`panel.cycle.print();`
`storage`	variable	total storage (bits)	`panel.storage.print();`
`storage_sram`	variable	total SRAM bits	`panel.storage_sram.print();`
`energy_fJ`	variable	total energy (femtojoules)	`panel.energy_fJ.print();`
`storage_xtors`	variable	total storage transistors	`panel.storage_xtors.print();`
`logic_xtors[0]`	variable	total logic transistors (0 = current cycle; 1 = previous)	`panel.logic_xtors[0].print();`
`logic_xtors[1]`	variable		
`total_xtors`	function	total transistors	`u64 x = panel.total_xtors();`
`dyn_power_mW`	function	dynamic power (milliwatt)	`f64 p = panel.dyn_power_mW();`
`sta_power_mW`	function	static power (milliwatt)	`f64 p = panel.sta_power_mW();`
`print`	function	print total	`panel.print();`
`next_cycle`	function	increment cycle	`panel.next_cycle();`

Table 4.1: Panel variables and functions for the user and the superuser. The user can read variables but cannot modify them. Function `next_cycle` is for the superuser only.

name	description	example
val/reg		
`get`	transform into C++ int	`val<4> x = 13;` `u64 v = x.get();`
`set_time`	set the timing (picoseconds)	`x.set_time(100);`
`time`	read the timing (picoseconds)	`u64 t = (x+1).time();`
`get_vt`	get both value and timing	`auto [v,t] = x.get_vt();`
arr		
`get`	transform into std::array of C++ int	`arr<val<4>,3> A = 1,2,3;` `std::array<u64,3> V = A.get();`
`set_time`	set the timing (same for all elements)	`A.set_time(100);`
`time`	maximum timing of elements	`u64 t = A.time();`

Table 4.2: Private functions that the superuser can use for interfacing with the rest of the simulator.

```
val<4> x = 13; // value=13, timing=0
val<4> y = {7,100}; // value=7, timing=100ps
```

The superuser has access to private assignment operators and can modify a `val` after construction.

4.3 The `next_cycle` function

The `next_cycle` function can be called only by the superuser. It is not considered part of the Harcom language but is nevertheless essential to the behavior of persistent types (regs and RAMs). The `next_cycle` function does three things:

- increment the cycle counter (variable `cycle`);
- save `logic_xtors[0]` into `logic_xtors[1]`;

option	effect
-DFREE_FANOUT	disables destructive reads and removes all read delays
-DCHECK_FANOUT	error at execution if actual fanout exceeds declared one
-DCHEATING_MODE	enables conversion of valtype to C++ int

Table 4.3: Compiler options for debugging.

- set `logic_xtors[0]` to zero.

Registers can be written only once per cycle; RAMs can be read and written only once per cycle. For example:

```
#include "harcom.hpp"
using namespace hcm;

struct harcom_superuser {
  reg<4> x = 0;

  void example() {
    x = x+1; // first write
    panel.next_cycle();
    x = x+1; // second write
    x.print();
  }
} hsu;

int main()
{
  hsu.example();
}
```

It is the call to `next_cycle` that makes the second write to `x` possible.[1]

4.4 Tips and suggestions

The constraints of the Harcom language allow to associate every statement with a hardware cost (which is null in certain cases). However, these constraints make the Harcom language less flexible than a general-purpose programming language. This section provides some tips and suggestions that users might find helpful for programming and debugging.

Table 4.3 lists the compiler options that are available for debugging purpose. The option `-DCHEATING_MODE` enables the conversion of valtypes to a C++ integer:

```
g++ -std=c++20 -o test_harcom test_harcom.cpp
     -Wall -Wextra -Werror -DCHEATING_MODE
```

For example, the user can introduce `assert` statements:

```
  val<4> x = 7;
#ifdef CHEATING_MODE
  assert(x==7);
#endif
```

[1]Otherwise, an error at execution is triggered.

The options -DFREE_FANOUT and -DCHECK_FANOUT are useful for debugging fanouts, as explained in Section 3.6.

There are two aspects to an algorithm written in the Harcom language: (1) the functional behavior of the algorithm and (2) its hardware complexity (timing, energy). While these two aspects are not completely independent of each other, the functional behavior is probably the aspect what we want to be correct first. During the initial development of an algorithm, it is not necessary to use the fanout and fo1 functions. Once the functional behavior is deemed correct, fanouts can be optimized. The -DFREE_FANOUT option can be used to obtain an upper bound of the delay that could be saved by optimizing fanouts. If the potential delay reduction is significant, the fanout function should be first applied to values with the highest fanout. Values with a lower fanout can be optimized if the potential delay reduction is still significant.

As explained in Section 3.6, function fo1 should be used very cautiously, as reading a value through it destroys the value. The -DFREE_FANOUT option can be used to check that the functional behavior is not altered by a misuse of fo1.

Programming in the Harcom language is actually programming in C++ with Harcom data types. For example, it is ok to have Harcom vals inside a C struct, a C array, or an std::array. An arr object can actually be constructed from a C array or an std::array:

```
val<4> AA[3] = {1,2,3};
arr<val<4>,3> A = AA;
std::array<val<4>,3> BB = {4,5,6};
arr<val<4>,3> B = BB;
```

However, Harcom data types are not guaranteed to work with popular containers or algorithms of the standard library. For example, if an std::tuple containing Harcom data types does not generate a compilation error, it probably means that the code is OK. However, there is no guarantee, and users should use the standard library very cautiously. Even if the code compiles, it is recommended to check the behavior.

Defining C++ classes containing Harcom data types is straightforward. Functions and function templates are straightforward too; for example:

```
template<u64 N>
val<N> max(val<N> x, val<N> y)
{
  x.fanout(hard<2>{});
  y.fanout(hard<2>{});
  return select(x>y,x,y);
}

template<u64 N, u64 M>
val<N> max(arr<val<N>,M> A)
{
  return fold(A.fo1(),max<N>);
}

int main()
{
  arr<val<5>,8> A = {8,13,2,5,19,4,23,10};
  max(A.fo1()).print();
}
```

Microarchitecture simulators are generally parameterized so that the simulated configuration can be changed easily. The sizes of all Harcom data types must be known at compile time. Writing parameterized components in the Harcom language can be done in two ways: (1) with preprocessor macros or (2) with C++ templates. We leave macros to the ingenuity of old-school C programmers and focus on templates here.

Template metaprogramming may be needed to compute certain template arguments at compile time. Template metaprogramming in C++20 is, fortunately, less cumbersome than in older (pre-C++11) versions of C++, thanks to the `constexpr` specifier, parameter packs and fold expressions.

The Harcom library provides a utility called `static_loop` that can be used for iterating over an integer template argument.

Chapter 5

Hardware complexity model

5.1 Limitations of the model

Bibliography

[1] CACTI. https://github.com/HewlettPackard/cacti.

[2] ChampSim. https://github.com/ChampSim/ChampSim.

[3] gem5. https://www.gem5.org/.

[4] McPAT. https://github.com/HewlettPackard/mcpat.

[5] J. Lowe-Power et al. The gem5 simulator: Version 20.0+. https://arxiv.org/abs/2007.03152, 2020.

[6] N. Gober, G. Chacon, L. Wang, P. V. Gratz, D. A. Jiménez, E. Teran, S. Pugsley, and J. Kim. The Championship Simulator: architectural simulation for education and competition. https://arxiv.org/abs/2210.14324, 2022.

[7] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. The McPAT framework for multicore and manycore architectures: simultaneously modeling power, area, and timing. *ACM Transactions on Architecture and Code Optimization*, 10(1), April 2013.

[8] C. Mead and L. Conway. *Introduction to VLSI systems*. Addison-Wesley, 1980.

[9] I. E. Sutherland, R. F. Sproull, and D. Harris. *Logical effort: designing fast CMOS circuits*. Morgan Kaufmann, 1999.

[10] S. J. E. Wilton and N. P. Jouppi. CACTI: an enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits*, 31(5), May 1996.