Harcom

Hardware complexity model for microarchitecture exploration

Pierre Michaud

June 16, 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 [7, 3] and ChampSim [5, 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 [8, 1] and McPAT [6, 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

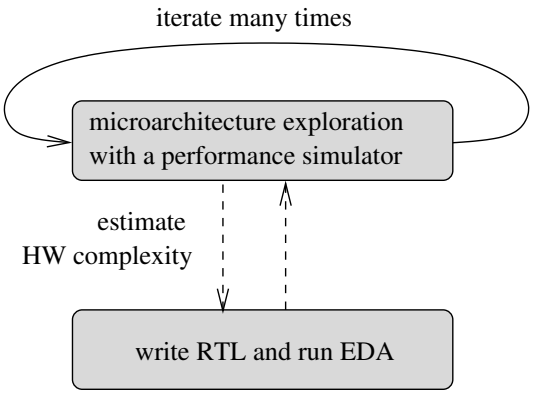

Figure 1.1: Hardware complexity estimation is off the main microarchitecture exploration loop.

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 Verilog or VHDL, 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 high-level synthesis tool. 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++ 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.[1]

Harcom's basic data type is called val. A val object is declared with a parameter N and represents an N-bit integer value[2] 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. **The timing of vals is just an output statistics, a measure of hardware complexity, it is not intended to be fed back to the performance simulator**.

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

[1] The simulator must be compiled with a recent compiler as Harcom is implemented in C++20.

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

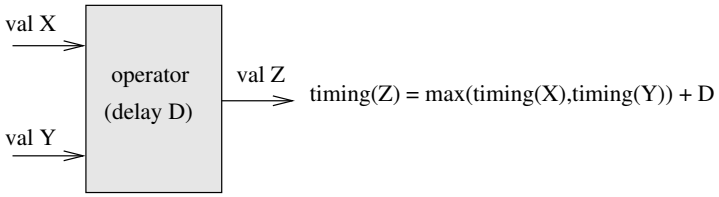

Figure 2.1: The timing of the result of a two-input operation is the maximum of the timing of the two inputs plus the latency of the hardware operator.

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

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

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
```

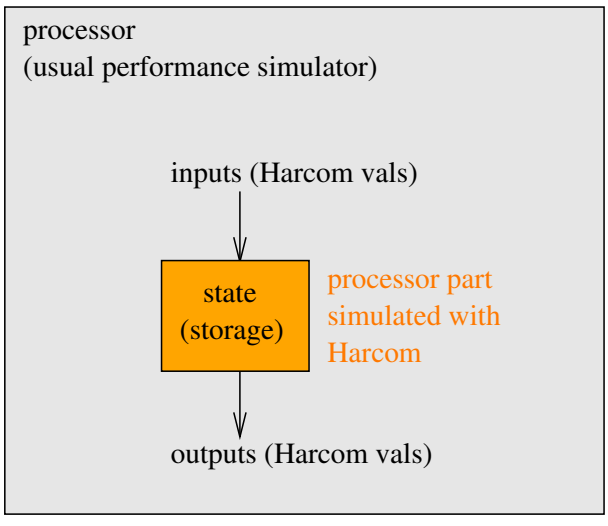

Figure 2.2: Harcom's typical usage: only the part of the performance simulator modeling the processor component that we want to study is rewritten.

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 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., is allowed to 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, with C++ as a metaprogramming language.

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]

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. For example:

[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.

```
x.print(); // print x in decimal (with the timing)
y.printb(); // print y in binary (with the timing)
// output z to the error stream, without the timing
z.print("z=","\n",false,std::cerr);
```

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]

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.

[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>>);)

3.1.4 The hard type

The hard type represents hardware parameters, that is, values that are fixed and known. 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 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:

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

3.2 Random access memory

3.3 Read only memory

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 3.1. Each operator takes one or two valtypes (val 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.

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

Table 3.1: Arithmetic/logical operators. Inputs are valtypes or hard values. Outputs are vals.

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] 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.

[6] 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.

[7] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jerónimo Castrillón, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Marjan Fariborz, Amin Farmahini Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc S. Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Éder F. Zulian. The gem5 simulator: Version 20.0+. https://arxiv.org/abs/2007.03152, 2020.

[8] 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.