Harcom

Hardware complexity model for microarchitecture exploration

Pierre Michaud

pierre.michaud@inria.fr

September 2, 2025

In memory of Stephan Jourdan (1971-2023)

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 [15, 3] and ChampSim [16, 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 [40, 1] and McPAT [25, 5]. 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, a timing in picoseconds and a location which can be 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 some global cost information such as the number of transistors, the energy consumption, etc.

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 global cost information (transistors, energy, storage, etc.), 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

---

[1]Future versions of Harcom might provide floating-point values.
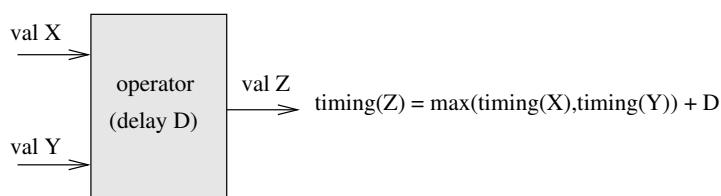
Figure 2.1: The timing of the result of a two-input operation is the maximum of the timing of the two inputs plus the latency of the hardware operator.

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

```cpp
#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=44 ps, loc=0)
//   storage (bits): 0
//   transistors: 406
//   fins/transistor: 1.69
//   SRAM area (mm2): 0
//   dynamic energy (fJ): 9.04
//   static power (mW): 0.000257
```
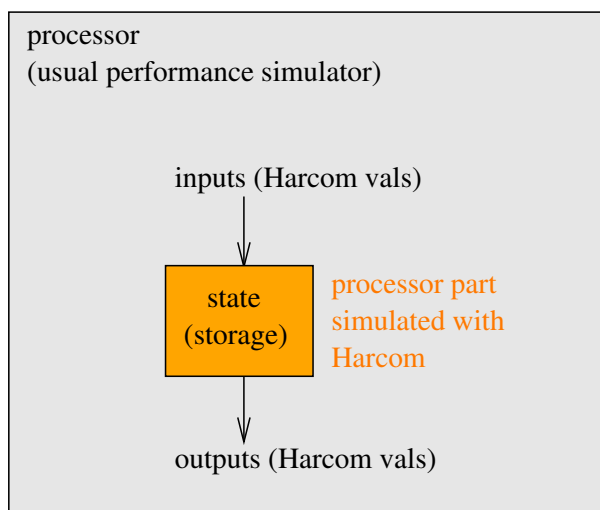

Figure 2.2: Harcom's typical usage: only the part of the performance simulator modeling the processor component that we want to study is rewritten.

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., can modify) the class called `harcom_superuser`. While the user is constrained by the Harcom language, the superuser can access private class 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.

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 (Section 3.6)	val<1> x = 1; x.fanout(hard<4>{});
fo1	function	set fanout of 1 (Section 3.6)	val<3> x = 1; val<3> y = x.fo1() + 1;

make_array	function	make an array from the value bits	val<12> x = 0b101011110011;
			arr<val<4>,3> 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();

connect	function	copy the val at a RAM location (Section 3.7)	ram<val<8>,64> M[4];
			panel.make_floorplan();
			val<8> x = M[0].read(0).connect(M[3]);

distribute	function	copy the val at RAM array locations (Section 3.7)	ram<val<8>,64> M[4];
			panel.make_floorplan();
			val<8> x = M[0].read(0);
			arr<val<8>,4> A = x.distribute(M);

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=4 ps, loc=0)"
y.printb(); // prints "1111 (t=8 ps, loc=0)"
```

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 exception at execution. 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. An arr can also be initialized from a C array or from 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 ;
```

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.

---

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

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

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;
```

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>{};
```

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(); // 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:

---

[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	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 (Section 3.6)	`A.fo1().concat().printb();`
`fo1`	function	set fanout of 1 (Section 3.6)	`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.

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`.

```
ram<arr<val<64>,2>,1024> mem;
val<10> addr = 100;
arr<val<64>,2> data = {addr,addr+1};
mem.write(addr,data);
mem.read(addr).print(); // 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 an exception is generated 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);};
```

---

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

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 (≤ 64 bits)
/	integer division	unsigned val dividend hard divisor	val with enough bits
%	modulo (remainder)		

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.

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` 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 Miscellaneous functions

Table 3.5 lists miscellaneous 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`. If F needs some captures, these must be by reference (`[&]`). 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;
```

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

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

split	split the bits of a val into multiple unsigned vals	`val<7> z = 0b1110011;`
		`auto [left,right] = split<3,4>(z);`
		`concat(left,right).printb();`

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	return the index of the set bit in a one-hot 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: Miscellaneous 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.

```
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, the operations *inside* F consume no energy and the storage (regs and RAMs) written by F is actually unmodified. Nevertheless, the transistor count is always incremented, regardless of the mask bit (the hardware is there, even if not used). Moreover, regardless of the mask bit, every attempt to write a reg or read/write a RAM 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[9]. Once the algorithm works as expected, the Harcom user tries to reduce the hardware cost.

Reading a `val` or a `reg` is associated with a delay and energy cost. 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 the delay and energy cost of a fanout-of-two (FO2) inverter. The read delay increases linearly with the number of reads.[10] 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 [27, 35], 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 exception at execution if the actual fanout exceeds the declared one.

3.6.2   The `fo1` function

Whenever possible,[11] 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.

---

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

[10]This corresponds to chaining FO2 inverters.

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

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.

3.7   Wiring cost

3.7.1   Floorplan

Harcom models, approximately, the silicon area occupied by RAMs. The function panel.make_floorplan() must be called when multiple RAMs are declared. This function generates a floorplan, each RAM being modeled as a rectangle. The floorplan can be visualized with the compiler option -DFLOORPLAN, which enables the generation of a Graphviz file floorplan.gv that can be processed with the dot command [4]. An optional label can be assigned to a RAM at declaration (see the example below), which helps identify the RAM in the floorplan visualization.

The floorplanning algorithm is somewhat primitive: all rectangles are sorted according to their areas, and rectangles are progressively combined to make larger rectangles.[12]

Each RAM corresponds to a particular location in space. For instance, if four RAMs are declared, then there are four possible locations, numbered 0,1,2,3 respectively. Each Harcom value, val or reg, has a location corresponding to one of the RAMs.

3.7.2   Location of a reg

The location of a reg is determined by its declaration order relative to RAMs. More specifically, a reg is located at the previous RAM, in declaration order, in the current region (see Section 3.7.6) or at the following RAM if there is no previous RAM:

```
reg<4> r0a; // location #0
ram<val<4>,64> m0 {"first␣RAM"}; //location #0
reg<4> r0b; // location #0
ram<val<4>,32> m1 {"second␣RAM"}; //location #1
reg<4> r1; // location #1
ram<val<4>,16> m2 {"third␣RAM"}; //location #2
reg<4> r2; // location #2
panel.make_floorplan();
r0a.print();
r0b.print();
r1.print();
r2.print();
```

3.7.3   Locus of an operation

The locus of a RAM operation (read or write) is the RAM itself. The *locus* of non-RAM operations (copy, arithmetic/logic, array folding, etc.) depends on the locations of valtype inputs. The locus of a non-RAM operation with a single valtype input is this input's location. The locus of a non-RAM operation with multiple valtype inputs is the location of the latest

---

[12]Some space is wasted when two rectangles of different dimensions are combined.

arriving input, that is, the input with the greatest timing. In general, the location of a val resulting from an operation is this operation's locus, with two exceptions (see Section 3.7.5).

3.7.4 Wiring cost

A *distant* input is an input whose location is different from the locus of the operation. Reading a distant input incurs a wiring cost, i.e., extra delay and energy. This cost increases with the wire length, which is the Manhattan distance between the input location and the operation locus. The `-DFREE_WIRING` compiler option removes the wiring cost and can be used to evaluate this cost.

3.7.5 Setting the location

Doing an operation at the location of the latest arriving input minimizes the wire delay penalty. However, we may want to do the operation at the location of the earliest arriving input instead if this reduces wire energy significantly Consider the following example:

```
ram<val<64>,16> m0; // fast RAM
ram<val<1>,1024> m1; // slow RAM
panel.make_floorplan();
val<64> x0 = m0.read(0);
val<1> x1 = m1.read(1);
(x0.fo1() & x1.fo1().connect(m0)).print();
```

The 64-bit value `x0` is read from fast RAM `m0`, the 1-bit value `x1` is read from slow RAM `m1`. By default, a 2-input operation on `x0` and `x1` is performed at `m1`, as `x1` arrives the latest. However, this implies reading a distant `x0`, which requires 64 wires. Instead, doing the operation at `m0` sligthly increases the overall delay but reduces energy significantly, as a single wire is needed to transport `x1`. We force the operation to be done at `m1` by moving the value with the `connect` function, which takes a RAM as argument (see Table 3.1).

When the same value is read from several distinct locations, Harcom models by default distinct wires from the value location to each destination location. However, this is generally suboptimal in terms of wire energy. The `distribute` function (see Table 3.1) can be used to send a value to an array of RAMs:

```
ram<val<64>,64> mem[16];
panel.make_floorplan();
val<64> data = 7;
arr<val<64>,16> d = data.distribute(mem);
for (u64 i=0; i<16; i++) {
  mem[i].write(0,d[i]);
}
panel.print();
```

The `distribute` function takes as argument either a C array or `std::array` of RAMs.

3.7.6 Regions

Distinct components of a microarchitecture that do not interact with each other, or interact infrequently, can be assigned to distinct *regions*. The Harcom user, and/or the superuser, can define such regions. The floorplanning algorithm keeps RAMs belonging to the same region close to each other. For example:

```
region R1;
ram<val<4>,64> m0;
ram<val<4>,32> m1;
region R2;
ram<val<4>,16> m2;
panel.make_floorplan();
R2.enter();
val<4> x = 1;
x.print();
panel.print(R1);
```

In this example, RAMs `m0` and `m1` are in region `R1` and RAM `m2` is in region `R2`.

Besides affecting the floorplan, regions allow to distinguish the global costs of distinct regions. The `panel.print()` function prints the global costs of all regions when called with no argument. When called with a region identifier, as in the example above, it prints the global costs of that specific region.

Regions also permit setting the location of vals (not regs) that are initialized with C++ litterals or variables. In the example above, the `R2.enter()` function call means that vals created from C++ literals/variables are located at the first RAM of region R2.[13]

If no region is defined explicity, an unnammed default region is created where all RAMs and registers are located.

3.7.7   Zones

Inside a region, it is possible to define *zones*. Zones only affect the floorplan. For example:

```
zone Z1;
ram<val<8>,64> m1[8] {"m1[0]"};
zone Z2;
ram<val<4>,32> m2[8] {"m2[0]"};
ram<val<8>,64> m3[8] {"m3[0]"};
panel.make_floorplan();
```

In the example above, RAM array `m1` is in zone `Z1` and and RAM arrays `m2` and `m3` are in zone `Z2`. Both zones belong to the default region (as no explicit region has been defined in this example). The floorplanning algorithm keeps arrays `m2` and `m3` close to each other.

3.8   The no-hidden-cost (NHC) rule

The Harcom language tries to take into account the hardware complexity of every statement. This is a general rule that can be called the *no-hidden-cost* rule, or **NHC**. Most violations of the NHC rule are caught at compile time with a compilation error, or at execution time with an exception and an error message.

However, unfortunately, not all violations of the NHC rule can be detected automatically. The Harcom language is not a proper programming language, it is C++ programming with Harcom types, plus some self-discipline from the programmer to not violate the NHC rule. In particular, using C++ integers is OK as long as this does not represent any hardware cost. For example, this is valid Harcom language:

---

[13]The `enter` function is useful mostly to the superuser for interfacing with Harcom components.

```
val<4> A[3] = {1,2,3};
reg<4> B[3];
for (int i=0; i<3; i++) // no hardware cost
  B[i] = A[i]; // hardware cost
```

In the example above, the body of the `for` loop has a hardware cost but the loop header itself bears no hardware cost, as the loop is equivalent to

```
B[0] = A[0];
B[1] = A[1];
B[2] = A[2];
```

Here are some guidelines to avoid violating the NHC rule:

- code written in the Harcom language must be located outside the `harcom_superuser` class;

- use valtype for data that is unknown at design time;

- do not use non-constant C++ integers (or any other fundamental type) whose lifetime spans multiple clock cycle; if you need a modifiable persistent variable, use a `reg`;

- do not access private class members;

- no type punning (C-style cast, `void*`, union, `reinterpret_cast`, `std::bit_cast`, `memcpy`, etc.)

- do not put Harcom types inside unions;

- no pointer to class member;

- compile with options `-Wall -Wextra -Werror`.

All features of the C++ language (array, class, function, template,...) can be used as long as the NHC rule is not violated. For example, consider the following two structs:

```
struct bundle1 {
  reg<4> x;
  int y; // NHC violation
};

struct bundle2 {
  reg<4> x;
  const int y; // ok
};
```

In the example above, the two structs contain a `reg` so they are implicitly persistent. However, the first struct violates the NHC rule because of the non-constant integer field, while the second struct is compatible with the Harcom language (provided the constness of `y` is not circumvented).

Harcom types are incompatible with many containers and algorithms of the C++ standard library. For example, trying to insert a `reg` inside an `std::vector` generates a compilation error. However, `std::array` and, to a certain extent, `std:tuple` are compatible with Harcom types. If using a class or function from the standard library generates no error at compilation and no exception at execution, this is probably OK. Nevertheless, even if everything seems OK,

rule	detected automatically?
data unknown at compile time must be a valtype	no
a `val` cannot be modified	yes
a `reg` can be modified only once per clock cycle	yes
all storage (regs, RAMs) must have the same lifetime	yes
a RAM can be read and written only once per cycle	yes
no access to private class members	to a certain extent
no Harcom type punning	no
no Harcom type in a union	no
the lifetime of a C++ integer must end before the next clock tick	no

Table 3.6: Main rules of the Harcom language

the standard library should not be used without doing tests to check that the NHC rule is not violated.

Table 3.6 summarizes the main rules of the Harcom language.

Chapter 4

Using the Harcom library

Figure 4.1 shows a contrived example of utilization of the Harcom library.

4.1 The panel

The *panel* is a global object.[1] The panel holds global costs such as number of transistors, energy, etc. For example, function `energy_fJ` gives the total energy (in femtojoules) dissipated so far:

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

Table 4.1 lists the panel variables and functions that the user and the superuser can utilize. Variable `clock_cycle_ps` has no impact on the execution, it is only used by Harcom for calculating dynamic power (`panel.dyn_power_mW`). It is set by default to 300 ps but can be modified by the superuser.

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 values 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 write or read the timing associated with a Harcom value. Table 4.2 lists the private functions of classes `val/reg` and `arr` that the superuser can use. The superuser has access to private assignment operators and can modify a `val` after construction.

---

[1]Harcom is not thread-safe: a single thread should execute all the Harcom code.

```
#include "harcom.hpp"

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

struct harcom_superuser {
  harcom_superuser() {
    hcm::panel.clock_cycle_ps = 200;
  }
  void one_cycle() {
    auto [v,t] = collatz(value).get_vt();
    assert(t < hcm::panel.clock_cycle_ps);
    hcm::panel.next_cycle();
    value = v;
  }
  uint64_t value = 27;
} 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`	function	total storage (bits)	`panel.storage().print();`
`storage_sram`	function	total SRAM bits	`panel.storage_sram().print();`
`energy_fJ`	function	total energy (femtojoules)	`panel.energy_fJ().print();`
`transistors`	function	total transistors	`panel.transistors().print();`
`xtor_fins`	function	total transistor fins	`panel.xtor_fins().print();`
`area_sram_mm2`	function	total SRAM area (mm^2)	`panel.area_sram_mm2().print();`
`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 global costs	`panel.print();`
`next_cycle`	function	increment cycle	`panel.next_cycle();`
`make_floorplan`	function	make the floorplan	`panel.make_floorplan();`

Table 4.1: Panel variables and functions. Functions can take a region identifier as argument, except `next_cycle` and `make_floorplan`. The user can read variables but cannot modify them. Function `next_cycle` is for the superuser only.

name	description	example
val/reg		
`get`	return a 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`	return a 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 (picoseconds), same for all elements	`A.set_time(100);`

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

option	effect
–DFREE_FANOUT	disables destructive reads and removes all read delays
–DCHECK_FANOUT	generate an exception if actual fanout exceeds declared one
–DCHEATING_MODE	enables conversion of valtype to C++ int
–DFLOORPLAN	`make_floorplan()` generates a `floorplan.gv` file
–DFREE_WIRING	removes wiring costs (except inside RAMs)

Table 4.3: Compiler options.

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 just increments the cycle counter (variable `cycle`). 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.[2]

4.4   Tips and suggestions

4.4.1   Compiler options

Table 4.3 lists compiler options. The option -DCHEATING_MODE enables the conversion of valtypes to C++ integers:

```
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;
```

---

[2]Otherwise, an exception is triggered.

```
#ifdef CHEATING_MODE
  assert(x==7);
#endif
```

The options -DFREE_FANOUT and -DCHECK_FANOUT were introduced in Section 3.6. The options -DFLOORPLAN and -DFREE_WIRING were introduced in Section 3.7.

All the source files that include the "harcom.hpp" header should use the same compiler options (One Definition Rule [6]).

4.4.2 Fanout and wiring

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, one may ignore fanout and wiring (sections 3.6 and 3.7). Once the functional behavior is deemed correct, fanout and wiring can be optimized. The -DFREE_FANOUT option can be used to obtain an upper bound of the delay that could be saved by optimizing fanout. The -DFREE_WIRING option can be used to obtain an upper bound of the delay and energy that could be saved by optimizing wiring.

As explained in Section 3.6, function fo1 should be used with caution as 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.

4.4.3 Separate Harcom-language code from the rest

The microarchitecture components whose hardware complexity we seek to estimate should be modeled with C++ functions written according to the rules of the Harcom language (Table 3.6), located outside the harcom_superuser class.

For example, in example 4.1, function collatz is written in the Harcom language. The persistent and modifiable variable value is not considered part of the component whose hardware complexity we seek to estimate, so we locate the variable outside of function collatz. Function collatz does not violate NHC as it does not access value directly but via a val

4.4.4 Templates

The Harcom library provides a utility function static_loop for iterating at compile time over an integer template argument. The function argument is a C++ lambda with one integer template parameter (C++20). For example, the following statement prints 0123456789:

```
static_loop<10>([]<int I>(){std::cout<<I;});
```

The static_loop utility simplifies template metaprogramming with Harcom types. For example:

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

template<u64... N>
struct regs {
  static constexpr u64 size = sizeof...(N);
```

```
  std::tuple<reg<N>...> tup;

  regs(auto... x) : tup{x...} {}

  void print() {
    static_loop<size>([&]<u64 I>(){
        std::get<I>(tup).print();
    });
  }
};

int main()
{
  regs<7,5,6> rs = {4,8,16};
  rs.print();
  panel.print();
}
```

Chapter 5

Hardware complexity model

There are many aspects to hardware complexity. Harcom considers three of them:

- number of transistors (and transistor fins);

- delay (combinational circuits and SRAM latency);

- energy/power.

It is possible to use Harcom without understanding the intricacies of its hardware complexity model. To know the hardware cost of a particular primitive of the Harcom language, it suffices to write a small test program and look at the numbers output by the library when the program is executed.

Nevertheless, one should not use a model without understanding its limitations. The purpose of this chapter is to help users make sense of the numbers output by Harcom. For an introduction to digital circuits, see [39].

5.1   Technology parameters and transistor model

All technology parameters of Harcom's complexity model, listed in Table 5.1, are taken or derived from various public sources. They are intended to model a plausible "5nm" FinFET technology, although there is no guarantee that they are realistic.[2]

We model a transistor as an RC circuit [39], as depicted in Figure 5.1. The effective resistance and gate capacitance are assumed identical for nFET and pFET ($\gamma = 1$). The gate and drain capacitances are assumed equal ($p_{inv} = 1$).

We consider two types of transistors: fast transistors for logic gates, and low-leakage transistors for SRAM cells.

We assume two types of metal wires, called Mx and My [29]. The Mx wires have a tight pitch and a high resistance per unit length. Harcom uses Mx wires only for wordlines and bitlines in SRAM banks [12]. All the other wires are My wires, which have twice the pitch of Mx ones and a much lower resistance per unit length [29].

---

[1]With the flying bitline technique, the top-half bitline has 65% of the capacitance of the conventional full bitline [12]. That is, $\frac{1+\delta/2}{1+\delta} = 0.65$, so $\delta = 0.35/0.15 \approx 2.33$.

[2]I did not have access to a commercial process development kit. Anyway, the use of a commercial PDK is generally constrained by a NDA, so technology parameters that are not disclosed by a foundry are at best educated guesses [37].

parameter	meaning	value	from
V_{dd}	supply voltage	0.75 V	[20, 12]
transistor model			
p_{inv}	ratio of drain capacitance to gate capacitance	1	[39]
C_g	gate capacitance per fin	0.0466 fF	$\delta c_w l_b / p_{inv}$
I_{dsat}	fast nFET saturation current per fin	60 μA	[37]
I_{off}	fast nFET leakage current per fin (25 °C)	1 nA	[37]
I_{dsat}^*	low-leakage nFET saturation current per fin	40 μA	[37]
I_{off}^*	low-leakage nFET leakage current per fin (25 °C)	17 pA	[37]
γ	pFET current relative to nFET	1	[17, 37]
I_{eff}	effective nFET drive current per fin	$I_{dsat}/2$	[33]
R_{eff}	effective resistance of single-fin nFET	$V_{dd}/(2I_{eff})$	[39]
τ	nFET intrinsic delay	$R_{eff}C_g$	[30]
wire model			
c_w	wire capacitance per unit length	0.2 fF/μm	[29, 20]
r_{wx}	Mx wire resistance per unit lentgh	150 Ω/μm	[29]
r_{wy}	My wire resistance per unit length	25 Ω/μm	[29]
6T SRAM cell (single fin)			
A	cell area ($l_w \times l_b$)	0.02 μm^2	[12]
ρ	cell aspect ratio (l_w/l_b)	2	[12]
l_w	cell wordline length	0.2 μm	$\sqrt{A\rho}$
l_b	cell bitline length	0.1 μm	$\sqrt{A/\rho}$
δ	nFET drain / bitline capacitance ($p_{inv}C_g/c_w l_b$)	2.33	[12][1]

Table 5.1: Parameters for a plausible "5nm" FinFET technology.
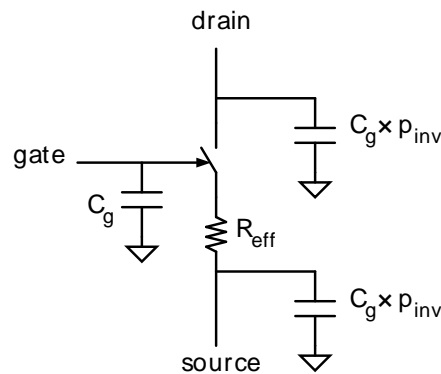
Figure 5.1: Simplified RC model of single-fin nFET. The pFET model is similar, with capacitors connected to V_{dd} instead of the ground.

5.2 Basic gates

The basic gates modeled in Harcom are: inverter, tristate inverter, NAND, NOR, XOR, AOI21 (AND-OR-invert), OAI21 (OR-AND-invert), and inverting MUX (multiplexer) [39]. While XOR-based circuits such as multipliers may benefit from pass-transistor logic [13], we assume static CMOS logic (i.e., complementary pull-down and pull-up networks) for all the gates, including XORs [41].

The transistors widths (number of fins) of a **unit-scale** gate are set so that the pull-down and pull-up networks each have worst-case resistance equal to R_{eff} (see Table 5.1). A scale-s gate can be obtained by multiplying the number of fins of each transistor of a unit-scale gate by the same factor s.

The 4-transistor (4T) tristate inverter has three inputs, D, E, and $\overline{E}$, where E is the enable input. The N-way inverting MUX consists of N tristate inverters with a common output.[3] The MUX select (one hot E) is not part of the MUX gate. The XOR gate ($A \oplus B$) has four inputs A, $\overline{A}$, B, $\overline{B}$ and is implemented like a 2-way multiplexer (8T) with $D_1 = A, E_1 = B, D_2 = \overline{A}, E_2 = \overline{B}$.

Basic gates in Harcom (class `basic_gate`) are represented with a few numbers, including number of transistors, input[4] capacitance c_i relative to C_g and output parasitic capacitance c_p relative to C_g. We estimate c_p as the sum of the drain capacitances of all the transistors that are attached directly to the gate output [35, 39].

5.2.1 Delay

The gate delay is modeled as

$$\text{gate delay} = \left(c_p + \frac{c_l}{s} \right) \times \tau \tag{5.1}$$

where τ is the intrinsic nFET delay, c_p is the output parasitic capacitance of the unit-scale gate relative to C_g, s is the gate's scale, and c_l is the load capacitance relative to C_g, which depends on the load that is connected at the gate output.

Delay increases with load capacitance. Increasing the scale s of G reduces G's delay. However, the (dimensionless) input capacitance is $s \times c_i$, hence scaling up G increases the previous gate's delay.

For example, applying Equation (5.1) to a fanout-of-k inverter (i.e., driving k identical inverters) gives

$$\begin{aligned} c_p &= p_{inv}(1 + \gamma) \\ c_l &= s \times k \times (1 + \gamma) \\ \text{FOk inverter delay} &= (1 + \gamma)(p_{inv} + k) \times \tau \\ &= 2(k + 1) \times \tau \end{aligned}$$

5.2.2 Energy

Dynamic energy dissipation mostly comes from charging and discharging capacitances.[5] A gate is *utilized* when its inputs have a non-null probability P_{sw} to switch.[6] The average dynamic

---

[3]In practice, $N \leq 4$.

[4]Asymmetric gates (tristate inverter, MUX, AOI21, OAI21) have multiple different c_i's.

[5]We neglect the energy dissipated by short-circuit currents that flow directly from V_{dd} to the ground while a gate is transitioning [39].

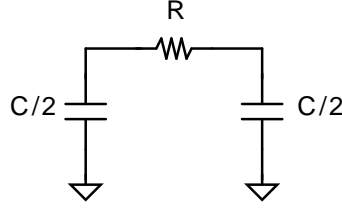[6]Glitches are ignored. They are not modeled.

Figure 5.2: Wire of total capacitance C and total resistance R modeled as a Π_1 network.

energy per gate utilization is

$$\text{energy} = P_{sw} \times \frac{1}{2} C_{sw} V_{dd}^2 \tag{5.2}$$

where C_{sw} is the total average switching capacitance of the gate. All inputs capacitances and certain drain/source capacitances contribute to C_{sw}. For an inverter, C_{sw} consists of two transistor gates and two transistor drains. However, for other gates (e.g., NAND), the contribution of drain/source capacitances depends on the probability $P_1 = 1 - P_0$ that an input is equal to 1. So C_{sw} depends on P_{sw} in the general case, as

$$P_{sw} = 2P_1(1 - P_1)$$

Even if we obtained a closed-form formula for C_{sw} (by solving a Markov chain), equation (5.2) would still be approximate because, in reality. different inputs may behave differently or be correlated. Moreover, the actual value of P_1 is unknown in practice. Instead, Harcom uses the following approximation:

$$C_{sw} = C_g(1 + p_{inv}) \times s \sum_{i=1}^{n_t} f_i \tag{5.3}$$

where s is the scale of the gate, n_t the number of transistors and f_i the number of fins per transistor in the unit-scale gate.

In general, Harcom does not know the actual value of P_1 and assumes

$$P_1 = 0.5$$

There are a few exceptions though. For instance, in a NAND-NOR tree [35], $\min(P_1, P_0)$ decreases geometrically with the depth of the gate in the tree. Harcom models this decrease.

5.3   Wires

The wires modeled in Harcom are unidirectional. Harcom is unaware of physical distances except in SRAMs. Therefore, wires are modeled in SRAMs only.

5.3.1   Delay

Wires impact gates delay by contributing to the load capacitance of gates. Wires also impact delay by their own electrical resistance. Harcom models a full-swing wire as a lumped Π_1 network [32], as depicted in Figure 5.2. Typically, such wire connects the output of a gate to the input of one or several other gates, which produces an RC tree network. Delay can be approximated as the first moment of the impulse response, which is known as the Elmore

delay.[7] The Elmore delay at node i in an RC tree is equal to $\sum_k R_{ki}C_k$ where k runs over all the nodes in the tree, C_k is the capacitance at node k and R_{ki} is the sum of resistances on the portion of path that is common to the path going from the input of the tree to node k and to the path going from the input to node i [34, 18]. As this formula is linear in both R and C, the delay contribution of each resistance can be computed separately (i.e, as if all the other resistances were null) and the overall delay is the sum of these contributions. We define the wire delay as the delay contribution of the wire resistance.

For a full-swing wire of total resistance R, total capacitance C, driving a load capacitance C_{load}, the wire delay is approximated as

$$\text{wire delay} = R \times (C/2 + C_{load}) \tag{5.4}$$

As R and C are both proportional to the wire length, the wire delay increases quadratically with length. Hence long My wires are split into segments, each segment but the first[8] being driven by a large inverter aka *repeater*. Equation (5.4) applies only to a segment. The total delay of the segmented wire is the sum of delays of each segment, which includes the repeater's delay.

The segment length that minimizes delay for an My wire is [39]:

$$L_{opt} = \sqrt{\frac{2(1+\gamma)(1+p_{inv})R_{eff}C_g}{r_{wy}c_w}} \tag{5.5}$$

The parameters in Table 5.1 yield $L_{opt} \approx 30.5\,\mu\text{m}$ for an My wire. Note that L_{opt} depends on R_{eff}, which depends on supply voltage V_{dd}.

In practice, the number of segments is an integer:

$$N_{seg} = \max(1, \lfloor L/L_{opt} + 0.5 \rfloor)$$

where L is the total length. The actual segment length is L/N_{seg}.

The repeater scale s that minimizes delay is [39]:

$$s = \sqrt{\frac{R_{eff}c_w}{(1+\gamma)r_{wy}C_g}} \tag{5.6}$$

Parameters in Table 5.1 give $s \approx 33$. When the total delay is minimized, the repeater's delay is equal to the segment wire's delay. The minimum segment delay (including the repeater) is

$$2(1+\gamma)\left(1 + p_{inv} + \sqrt{2(1+p_{inv})}\right) \times \tau = 16\tau$$

where $\tau = R_{eff}C_g = 0.5825\,\text{ps}$ (Table 5.1).

The repeater's switching capacitance is

$$s \times (1+\gamma)(1+p_{inv})C_g = L_{opt}c_w \times \sqrt{(1+p_{inv})/2}$$

That is, when delay is minimized, the repeater's switching capacitance is equal to the segment wire capacitance ($p_{inv} = 1$).

---

[7]The Elmore delay is an accurate estimate of the 50% delay when the input voltage is a slow ramp, otherwise it is an overestimate [18, 39].

[8]The first segment is driven by a tapered buffer to present a small input capacitance to the gate driving the wire.

5.3.2   Energy

When a wire of capacitance C experiences a charging corresponding to a voltage increase $\Delta V > 0$ and then a discharging $-\Delta V$ (i.e., going back to its initial voltage), the switching energy dissipated by Joule heating is

$$E(\Delta V) = C \times \Delta V \times V_{dd} \qquad\qquad (5.7)$$

This corresponds to the difference of potential energy of a positive charge $C\Delta V$ taken from the electric potential V_{dd} and brought to ground potential zero. The energy dissipated by a discharging $-\Delta V$ followed by a charging ΔV is the same. For a full-swing wire, i.e., $\Delta V = V_{dd}$, the charging and discharging dissipate the same energy, which is $CV_{dd}^2/2$. All wire modeled in Harcom are full-swing wires except SRAM bitlines on reads.

Note that about half of the switching energy of a segmented wire comes from repeaters. Here, Harcom prioritizes delay over energy.[9]

5.4   Combinational circuits

A combinational circuit (class `circuit` in "harcom.hpp") is modeled with six parameters, the four most important ones being the number of transistors, the input capacitance, the delay, and the energy. Combinational circuits can be obtained from basic gates or by combining smaller combinational circuits. Combinational circuits in SRAM periphery can contain wires.

The energy of a circuit is the sum of the energies of its components. The delay of a circuit is the worst-case propagation delay between any input and any output.

In certain circuits,[10] the gates are sized to minimize the circuit delay [35]. Other circuits are not optimized with respect to gate sizing. Instead, they use unit-scale gates and tapered buffers for driving large capacitances.[11]

Each operation of the Harcom language with a non-null hardware cost is associated with a subcircuit. The subcircuit parameters are calculated at compile time. They are used at execution time to obtain the timing of output values and to update transistor and energy counters.

The subcircuits modeled in Harcom are not necessarily the best possible implementations. For instance, Harcom's full adder has 34 transistors, its delay is 2.6 FO4 inverter delays, and its input capacitance $14C_g$. It is certainly possible to do better [39]. On the other hand, Harcom does not model wires outside of SRAMs, which means that, for subcircuits with much wiring such as the Kogge-Stone adder [39, 19], Harcom underestimates the actual delay and energy.

ROMs (see Section 3.3) are modeled as combinational logic, that is, with a decoder and multiple OR trees. This model is OK for small ROMs but is not realistic for large ROMs.

5.5   SRAM (static RAM)

Harcom models a single type of SRAM: high-density 6T SRAM cell (single read/write port). A high-density cell contains six single-fin transistors [12, 22]: two cross-coupled unit-scale inverters and two access nFETs. Wordlines and bitlines use Mx wires. The peripheral circuitry use My wires.

---

[9]Future versions of Harcom may trade a longer delay for lower energy by using longer segments and/or smaller repeaters.

[10]More specifically: tapered buffer, NAND-NOR tree, 2-input XOR, majority, half adder and full adder.

[11]A tapered buffer is chain of inverters of geometrically increasing scale, starting from a unit-scale inverter. The scale ratio β between successive inverters is a tradeoff between delay and energy [14].

5.5.1 SRAM bank

Small SRAMs consist of a single bank, large SRAMs consist of multiple banks. A bank possesses its own decoder and circuitry for reading/writing data. A bank is characterized by three parameters, N, M, D: N is the number of wordlines (number of rows), M the number of cells per wordline (number of columns) and D the width in bits of the accessed data. M must be a multiple of D ($D \leq M$) and M/D a power of two. Each cell is at the intersection of a row and a column. There are M columns, hence $2M$ bitlines.

Each wordline and each bitline uses an Mx wire. The bank array is physically split into a left half and a right half. The decoder (see [39, 9]) is located between the two halves and drives the wordline from the midpoint.

Harcom assumes small-signal bitline sensing [39, 22]. When $M > D$, column multiplexing is done after bitline sensing.[12] Hence there is one sense amplifier (SA) per column.

Bitlines are precharged to V_{dd} before any access (read or write). The precharge time is not modeled; that is, precharge is instantaneous after the access. The precharge circuitry is not modeled either.

A maximum gate scale of 10 is assumed in the wordline drivers and bitline write drivers, because of area constraints. This impacts decode delay and write latency.

Allowing one SRAM access per cycle implies pipelining (for large SRAMs). Pipelining is assume to be perfect: the circuitry necessary for pipelining (registers, clock, etc.) is not modeled. Harcom does not model the SRAM bank cycle (read/write + precharge) and therefore cannot check that it is compatible with `clock_cycle_ps` (Table 4.1). It is assumed that they are compatible.

Flying bitlines reduce the bitline capacitance [10, 12, 22]. However, this is not modeled in Harcom yet.

Wordline delay. The wordline delay is obtained from equation (5.4):

$$
\begin{aligned}
R &= r_{wx}l_w \times M/2 \\
C &= (c_w l_w + 2C_g) \times M/2 \\
C_{load} &= 0 \\
\text{wordline delay} &= r_{wx}l_w(c_w l_w + 2C_g) \times M^2/8
\end{aligned}
$$

The wordline delay increases quadratically with M, so M is limited in practice.

Bitline delay. On a read, N bitlines discharge slowly, until the voltage drop $-\Delta V$ is sufficient to be detected by the SA. The SA is a latch consisting of two cross-coupled inverters and isolation transistors [8, 21]. Harcom's model assumes that the voltage drop $-\Delta V$ that the SA can detect is inversely proportional to the square root of the SA scale [7, 22]:

$$
\Delta V = \sqrt{\frac{10}{s_{sa}}} \times 0.1 \text{ volt} \tag{5.8}
$$

where $s_{sa} \leq 10$ is the SA scale.[13] Enlarging the SA makes it less sensitive to random variations, which reduces its offset voltage [31, 23].

---

[12]Real SRAMs do column multiplexing before bitline sensing, which reduces the number of sense amplifiers and the read energy. This will need to be corrected in future versions of Harcom.

[13]Unfortunately, factors 0.1 and 10 are random numbers. Vashishtha et al. assume $\Delta V = 100\,\text{mV}$, while Mathur et al. assume $\Delta V = 150\,\text{mV}$ [38, 26]. Neither paper provides the SA size.

Equation (5.4) cannot be used to obtain the bitline delay under small-signal sensing. As explained by Amrutur and Horowitz, the bitline delay is better modeled by considering the SRAM cell as a current source [8]. We assume that the cell draw a constant current I_{dsat}^*. Let R and C be respectively the total bitline resistance and total bitline capacitance (including the drain capacitances of N access nFETs), and C_{sa} the SA input capacitance:

$$
\begin{aligned}
R &= r_{wx}l_b \times N \\
C &= (c_w l_b + p_{inv}C_g) \times N \\
C_{sa} &= (1+\gamma)(1+p_{inv})C_g \times s_{sa}
\end{aligned}
$$

The Amrutur-Horowitz formula can be applied to a distributed RC line [28], which yields the following approximate bitline delay:

$$
\text{bitline delay} = \frac{RC}{6} \times \frac{1+3C_{sa}/C}{1+C_{sa}/C} + \frac{(C+C_{sa})\Delta V}{I_{dsat}^*}
$$

Bitline delay is the sum of two terms. The first term increases with C_{sa} (but remains bounded by $RC/2$). The second term depends on ΔV and is minimized when $C_{sa} = C$ (owing to the square root in equation (5.8)). Unless N is excessively large, the second term dominates, and the tradeoff between the SA's capacitance and its offset voltage is at work [22]. Harcom sets s_{sa} as

$$
s_{sa} = \max\left(1, \min\left(10, \frac{0.4 \times C}{(1+\gamma)(1+p_{inv})C_g}\right)\right)
$$

The 0.4 coefficient is a tradeoff between energy and delay.

Energy   Wordlines consume little energy as only a single wordline switches per access. Bitlines consume a significant part of the energy per access. Harcom optimistically assumes that, on a read access, the bitline voltage swing is ΔV, the amount that the SA can detect reliably.[14] So the bitline energy is small, as per equation (5.7). However, M bitlines experience the voltage swing, which means that some energy is wasted when $M > D$. On a write access, bitlines experience a full voltage swing, however only D bitlines are concerned.

5.5.2   Multi-bank array

The wordline RC delay increases quadratically with M. The bitline delay increases mostly linearly with N (until N becomes very large, when the quadratic component is no longer negligible). Even if linear, this is a fast increase owing to the small cell current.

N and M must be large enough to allow high SRAM density and low energy. However, N and M are limited in practice, and a realistic SRAM bank cannot exceed a certain size. Therefore, large SRAMs are split into multiple banks.

The SRAM is a rectangular $N_x \times N_y$ array of banks. The accessed data lies in a single bank if its size does not exceed M bits. Otherwise the data is split into chunks that go to several adjacent banks along the x direction.

The routing of address and data between the SRAM access point and the banks is done with H-tree networks [36]. All interconnects between banks are unidirectional. There are three separate H-trees for address, read data and write data.

The address and write-data H-trees transport packets consisting of payload bits (the local bank address or the write data) and some bank-ID bits. When a packet arrives at a fork, one

---

[14]Amrutur and Horowitz assume that the voltage swing is twice ΔV [8].

bank-ID bit is used to route the packet to one of two path. After the packet has passed a fork, the bank-ID is one bit shorter.

For reads, the bank (or group of banks) containing the accessed data is selected first. Bank selection is done with N_x vertical wires and N_y horizontal wires, driven by two decoders. A bank select signal is generated at each bank by ANDing the horizontal and vertical select wires. Bank selection is done in parallel with the routing of the local bank address on the address H-tree. The bank read (for a selected bank) starts after the bank select signal has been generated and the local bank address has arrived. The data read from the bank is sent into the read-data H-tree, along with a tag consisting of the bank select bit. The read-data H-tree consist of a tree of two-way MUXes and an OR tree. The two trees are intermingled. At each node, each of the two input tags controls one of the two tristate inverters making the MUX. By construction, at most one the two input tags is set to one. The output tag is obtained by ORing the two input tags. While the bank is being read, the bank select bits are injected into the OR tree. The data read from the bank is injected into the MUX tree and travels up to the SRAM entry point.

Energy. Only the banks containing the accessed data consume dynamic energy. As for the H-trees, it is assumed that all the banks have equal probability to be accessed. For the address and write-data H-trees, the part of the H-tree that is not selected is assumed to not switch and does not dissipate dynamic energy.[15] Similarly, for the read-data H-tree, the wires that are off the path from the accessed bank to the entry point do not switch and do not dissipate dynamic energy.

5.5.3 Array configuration

The SRAM latency and energy depends on parameters N_x, N_y, N, M and D. Harcom considers multiple configurations and selects the one deemed the best according to a cost function:

$$\text{cost} = (E_r \times 2 + E_w) \times T_r^3$$

where E_r and E_w are respectively the read and write energy per access and T_r is the SRAM read latency. This cost function prioritizes read latency over energy, and read energy over write energy. As a result, it can happen that reducing the size of a SRAM increases its energy slightly. Although such non-monotonicity is undesirable, it is difficult to get rid of non-monotonic behaviors completely when considering simultaneously latency and energy.

As the exploration is done at compile time, Harcom evaluates only few configurations. Harcom prunes the configuration space by considering banks and arrays whose aspect ratio is squarish, to the extent possible.

Although the SRAM write latency is not used by Harcom, future versions of Harcom will need the cost function to take into account write latency, as this impacts write energy.

5.5.4 The `sram<E,D>` type

The Harcom user can access the hardware models directly. For example, the following command prints the main characteristics (latency, energy,...) of a 4096×16 SRAM, that is a 64-Kbit SRAM with 16-bit data size:

```
sram <4096 ,16>:: print ();
```

---

[15]Forks are implemented with tristate buffers. The unselected path is left floating. It is assumed that glitches can be prevented.

More detailed information can be obtained:

```
sram <4096 ,16 >:: print2 (); // prints Bx, By, N, M
sram <4096 ,16 >:: ARR :: print2 (); // array information
sram <4096 ,16 >:: ARR :: BANK :: print2 (); // bank detail
```

The two parameters E (for entries) and D for (data) of the sram<E,D> type are not required to be power-of-two values. However, the actual number of SRAM bits may be slightly greater than E×D because two of the three parameters N, N_x and N_y must be power-of-two values.

The static_loop utility introduced in Section 4.4.4 can be used to see how latency and energy varies with E and D. For example:

```
static_loop <5>([] <u64 I >(){
    sram <(1024 < <I) ,16 >:: print ();
});
```

5.6   Leakage power

Harcom assumes two types of transistors: low-leakage transistors in SRAM cells and fast transistors everywhere else. Let F be the total number of transistor fins and S the total declared SRAM capacity in bits. Leakage power is modeled as

$$P_{leak} = \left(3S \times I_{off}^* + \frac{1}{2}(F - 6S) \times I_{off} \right) \times V_{dd} \tag{5.9}$$

where I_{off} (and I_{off}^*) is the nFET subthreshold leakage current per fin at 25 °C when $V_{gs} = 0$ and $V_{ds} = V_{dd}$. Note that each SRAM cell has 6 single-fin transistors (Table 5.1).

Formula (5.9) is a very rough model. Harcom ignores the following facts:

- **leakage underestimated:**

  - leakage current increases with temperature; 25 °C is not a realistic temperature for high-performance processors.

- **leakage overestimated:**

  - the stack effect (transistors in serie) reduces leakage [39, 11];

  - fast transistors are used only in circuit paths that are critical for delay;

  - combinational circuits that are used infrequently are power gated.

Future versions of Harcom will need a more accurate leakage power model (see [11] for instance).

5.7   User-defined fanout

Each primitive of the Harcom language with a hardware cost is associated with a subcircuit. However, Harcom does not have a global view of how the multiple subcircuits declared by the user are interconnected. In particular, Harcom does not know the type and number of gates to which the output of a subcircuit is connected. Harcom optimistically assumes that the output of a subcircuit is connected to the input of one or several unit-scale inverters. Here, Harcom

understimates the output capacitance and therefore the delay of the subcircuit, as the input capacitance of a subcircuit (e.g., a full adder) can be greater than that of an inverter.[16]

Nevertheless, Harcom takes into account the fanout by attributing a hardware cost to reading values. If the `fanout` primitive is not used, Harcom assumes that each use of the output of a subcircuit costs a FO2 inverter, which leads to a linear chain of FO2 inverters (Section 3.6). This artificial hardware cost is an incentive to use the `fanout` primitive. When used properly, the `fanout` primitive makes the delay logarithmic instead of linear. The subcircuit associated with the `fanout` primitive is a tree of FO4 inverters. Even a fanout as small as FO2 must have a non-null hardware cost, otherwise it would be possible to replicate a value, then replicate the copies, and so on, at no hardware cost, which would be unrealistic. In a real circuit, the actual fanout is known at design time, and a tapered buffer would be used instead of an FO4 tree. Nevertheless, the delay and energy of an FO4 tree is roughly equivalent to that of a tapered buffer with scaling factor $\beta = 4$.

5.8 Floorplan and wiring cost

Distant locations (see Section 3.7) are connected by My wires. There is no limitation on the number of wires between two locations.

On the one hand, Harcom models (approximately) the area of SRAMs but does not model the area of logic circuits and registers. This means that a substantial part of the wiring is not modeled, and, for the part that is modeled, physical distances might be underestimated. On the other hand, the floorplanning algorithm used by Harcom is rudimentary, which leads to overestimating physical distances. So the modeling of wiring costs by Harcom is a rough estimation.

5.9 Limitations of Harcom

Below is a non-exhaustive list of sources of inaccuracy in Harcom (in no particular order):

- the transistor model is ultra simplified;

- the technology parameters of Table 5.1 are guesses;

- only two types of wires are modeled;

- the area of registers and combinational circuits is not modeled;

- wires in combinational circuits are not modeled, yet their cost is important [24];

- a single type of SRAM is modeled, which prioritizes density;

- the SRAM is modeled approximately;

- only a few SRAM configurations are explored, and not all parameters are varied; the best configuration identified by Harcom is probably suboptimal; write latency is ignored in selecting the "best" configuration;

- logic circuits use a single type of transistor;

---

[16]The output capacitance is also underestimated because wiring is ignored but in SRAMs.

- voltage is constant (dynamic voltage scaling is not modeled);

- true activity factors are unknown (assume $P_{sm} = 0.5$ is most situations);

- glitching power and short-circuit power are ignored;

- Harcom registers are not clocked; the clock is not modeled;

- registers necessary for pipelining SRAMs or combinational circuits are not modeled;

- the user can create circuits only from the available primitives; for instance, it is not possible to implement a mirror adder from scratch [39]; there is no primitive permitting non-CMOS logic or analog circuits; the user cannot set the number of fins per transistor;

Acknowledgment

Bibliography

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

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

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

[4] Graphviz. https://en.wikipedia.org/wiki/Graphviz.

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

[6] One Definition Rule. https://en.wikipedia.org/wiki/One_Definition_Rule.

[7] M. H. Abu-Rahma, Y. Chen, W. Sy, W. L. Ong, L. Y. Ting, S. S. Yoon, M. Han, and E. Terzioglu. Characterization of SRAM sense amplifier input offset for yield prediction in 28nm CMOS. In *Custom Integrated Circuits Conference (CICC)*, 2011.

[8] B. S. Amrutur and M. A. Horowitz. Speed and power scaling of SRAM's. *IEEE Journal of Solid-State Circuits*, 35(2), February 2000.

[9] B. S. Amrutur and M. A. Horowitz. Fast low-power decoders for RAMs. *IEEE Journal of Solid-State Circuits*, 36(10), October 2001.

[10] J. Chang andY. H. Chen, W.-M. Chan, S. P. Singh, H. Cheng, H. Fujiwara, J.-Y. Lin, K.-C. Lin, J. Hung, R. Lee, H.-J. Liao, J.-J. Liaw, Q. Li, C.-Y. Lin, M.-C. Chiang, and S.-Y. Wu. A 7nm 256Mb SRAM in high-K metal-gate FinFET technology with write-assist circuitry for low-Vmin applications. In *International Solid-State Circuits Conference (ISSCC)*, 2017.

[11] J. A. Butts and G. S. Sohi. A static power model for architects. In *International Symposium on Microarchitecture (MICRO)*, 2000.

[12] T.-Y. J. Chang, Y.-H. Chen, W.-M. Chan, H. Cheng, P.-S. Wang, Y. Lin, H. Fujiwara, R. Lee, H.-J. Liao, P.-W. Wang, G. Yeap, and Q. Li. A 5-nm 135-Mb SRAM in EUV and high-mobility channel FinFET technology with metal coupling and charge-sharing write-assist circuitry schemes for high-density and low-Vmin applications. *IEEE Journal of Solid-State Circuits*, 56(1), January 2021.

[13] A. L. Chinazzo, J. Lappas, C. Weis, Q. Huang, Z. Wu, L. Ni, and N. Wehn. Investigation of pass transistor logic in a 12nm FinFET CMOS technology. In *International Conference on Electronics, Circuits and Systems (ICECS)*, 2022.

[14] J.-S. Choi and K. Lee. Design of CMOS tapered buffer for minimum power-delay product. *IEEE Journal of Solid-State Circuits*, 29(9), September 1994.

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

[16] 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.

[17] X. Guo, V. Verma, P. Gonzalez-Guerrero, S. Mosanu, and M. R. Stan. Back to the future: digital circuit design in the FinFET era. *Journal of Low Power Electronics*, 13(3), September 2017.

[18] R. Gupta, B. Tutuianu, and L. T. Pileggi. The Elmore delay as a bound for RC trees with generalized input signals. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(1), January 1997.

[19] D. Harris and I. Sutherland. Logical effort of carry propagate adders. In *Asilomar Conference on Signals, Systems and Computers*, 2003.

[20] IRDS. International Roadmap for Devices and Systems - More Moore. https://irds.ieee.org/editions/2021, 2021.

[21] H. Jeong, J. Park, T. W. Oh, W. Rim, T. Song, G. Kim, H.-S. Won, and S.-O. Jung. Bitline precharging and preamplifying switching pMOS for high-speed low-power SRAM. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 63(11), November 2016.

[22] Y. Kim, C. Ong, A. M. Pillai, H. Jagadeesh, G. Baek, I. Rajwani, Z. Guo, and E. Karl. Energy-efficient high bandwidth 6T SRAM design on Intel 4 CMOS technology. *IEEE Journal of Solid-State Circuits*, 58(4), April 2023.

[23] K. J. Kuhn, M. D. Giles, D. Becher, P. Kolar, A. Kornfeld, R. Kotlyar, S. T. Ma, A. Maheshwari, and S. Mudanai. Process technology variation. *IEEE Transactions on Electron Devices*, 58(8), August 2011.

[24] Y. Lee, W. Jang, K. Kwon, J. Park, C. Yoo, J. Chang, and J. Jeon. Investigation on the effects of interconnect RC in 3nm technology node using path-finding process design kit. *IEEE Access*, 10, August 2022.

[25] 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.

[26] R. Mathur, M. Bhargava, S. Salahuddin, P. Schuddinck, J. Ryckaert, S. Annamalai, A. Gupta, Y. K. Chong, S. Sinha, B. Cline, and J. P. Kulkarni. Buried bitline for sub-5nm SRAM design. In *IEEE International Electron Devices Meeting (IEDM)*, 2020.

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

[28] P. Michaud. Effect of the wire resistance on the bitline read delay in SRAMs, October 2024. https://files.inria.fr/pacap/michaud/rc_delay.pdf.

[29] S. Nikolić, F. Catthoor, Z. Tőkei, and P. Ienne. Global is the new local: FPGA architecture at 5nm and beyond. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2021.

[30] E. J. Nowak. Maintaining the benefits of CMOS scaling when scaling bogs down. *IBM Journal of Research and Development*, 46(2/3), March 2002.

[31] M. J. M. Pelgrom, H. P. Tuinhout, and M. Vertregt. Transistor matching in analog CMOS applications. In *International Electron Devices Meeting (IEDM)*, 1998.

[32] V. B. Rao. Delay analysis of the distributed RC line. In *Design Automation Conference (DAC)*, 1995.

[33] A. Razavieh, Y. Deng, P. Zeitzoff, M. H. Na, J. Frougier, G. Karve, D. E. Brown, T. Yamashita, and E. J. Nowak. Effective drive current in scaled FinFET and NSFET CMOS inverters. In *Device Research Conference (DRC)*, 2018.

[34] J. Rubinstein, P. Penfield Jr., and M. A. Horowitz. Signal delay in RC tree networks. *IEEE Transactions on Computer-Aided Design*, CAD-2(3), July 1983.

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

[36] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi. CACTI 5.1. Technical Report HPL-2008-20, HP Labs, April 2008.

[37] V. Vashishtha and L. T. Clark. Comparing bulk-Si FinFET and gate-all-around FETs for the 5 nm technology node. *Microelectronics Journal*, 126, August 2022.

[38] V. Vashishtha, M. Vangala, P. Sharma, and L. T. Clark. Robust 7-nm SRAM design on a predictive PDK. In *International Symposium on Circuits and Systems (ISCAS)*, 2017.

[39] N. H. E. Weste and D. M. Harris. *CMOS VLSI design: a circuits and systems perspective*. Addison-Wesley, fourth edition, 2010.

[40] 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.

[41] R. Zimmermann and W. Fichtner. Low-power logic styles: CMOS versus pass-transistor logic. *IEEE Journal of Solid-State Circuits*, 32(7), July 1997.