

3. Sequential Logic ¹

“It’s a poor sort of memory that only works backward.”

Lewis Carroll (1832-1898)

All the Boolean and arithmetic chips that we built in previous chapters were *combinational*. Combinational chips compute functions that depend solely on *combinations* of their input values. These relatively simple chips provide many important processing functions (like the ALU), but they cannot *maintain state*. Since computers must be able to not only compute values but also to store and recall values, they must be equipped with memory elements that can preserve data over time. These memory elements are built from *sequential chips*.

The implementation of memory elements is an intricate art involving synchronization, clocking, and feedback loops. Conveniently, most of this complexity can be embedded in the operating logic of very low-level sequential gates called *flip-flops*. Using these flip-flops as elementary building blocks, we will specify and build all the memory devices employed by typical modern computers, from binary cells to registers to memory banks and counters. This effort will complete the construction of the chip-set needed to build an entire computer – a challenge that we take up in the next chapter.

Following a brief overview of clocks and flip-flops, section 1 introduces all the memory chips that we will build on top of them. Sections 2 and 3 describe the chips specifications and implementation, respectively. As usual, all the chips mentioned in the chapter can be built and tested using the hardware simulator supplied with the book.

1. Background

The act of “remembering something” is inherently a function of time: you remember *now* what has been committed to memory *before*. Thus, in order to employ chips that “remember” information, we must first develop some means for representing the progression of time.

The Clock: In most computers, the passage of time is marked by a master clock that delivers a continuous train of alternating signals. The exact hardware implementation is usually based on an oscillator that alternates continuously between two phases labeled “0-1”, “*low-high*”, “*tick-tock*”, etc. The elapsed time between the beginning of a “tick” and the end of the subsequent “tock” is called *cycle*, and each clock cycle is treated as a discrete time unit. The current clock phase (*tick* or *tock*) is represented by a binary signal. Using the hardware’s circuitry, this signal is simultaneously broadcast to every sequential chip throughout the computer platform.

Flip-flops: The most elementary sequential element in the computer is a device called *flip-flop*, of which there are several variants. In this book we use a variant called *data flip-flop*, or DFF, whose interface consists of a single-bit data input and a single-bit data output. In addition, the DFF has a *clock* input that continuously changes according to the master clock’s signal. Taken

¹ From *The Elements of Computing Systems*, Nisan & Schocken, MIT Press, forthcoming in 2003, www.idc.ac.il/csd

together, the data and the clock inputs enable the DFF to implement the time-based behavior $out(t) = in(t-1)$, where in and out are the gate's input and output values and t is the current clock cycle. In other words, the DFF simply outputs the input value from the previous time unit.

As we now turn to show, this elementary behavior can form the basis of all the hardware devices in the computer that have to *maintain state*, from binary cells to registers to arbitrarily large random access memory units.

Registers: A *register* is a storage device that can “store,” or “remember,” a value over time, implementing the classical storage behavior $out(t) = out(t-1)$. A DFF, on the other hand, can only output its previous input, i.e. $out(t) = in(t-1)$. This suggests that a register can be implemented from a DFF by simply feeding the DFF output back into its input, creating the device shown in the middle of Fig. 1. Presumably, the output of this device at any time t will equal its output at time $t-1$, yielding the classical function expected from a storage unit.

Well, not so. The device shown in the middle of Fig. 1 is invalid. First, it is not clear how we'll be able to ever load this device with a new data value, since there are no means to tell the DFF when to draw its input from the in wire and when from the out wire. More generally, the rules of chip design dictate that internal pins must have a fan-in of 1, meaning that they can be fed from a single source only.

The good thing about this thought experiment is that it leads us to the correct and elegant solution shown in the right of Fig. 1. In particular, a natural way to resolve our input ambiguity is to introduce a multiplexor into the design. Further, the “select bit” of this multiplexor can become the “load bit” of the overall register chip: if we want the register to start storing a new value, we can put this value in the in input and set the $load$ bit to 1; if we want the register to keep storing its internal value until further notice, we can set the $load$ bit to 0.

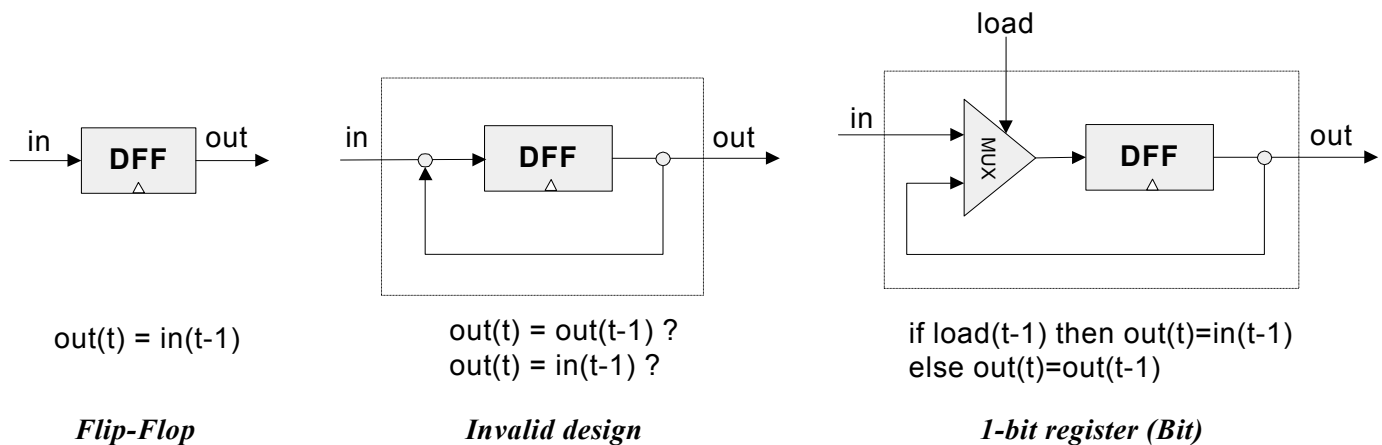


FIGURE 1: From DFF to single-bit register. The small triangle represents the clock input of the DFF. In chip diagrams, this icon states that the marked chip, as well as the overall chip that encapsulates it, are time-dependent.

Once we have the basic ability to remember a single bit over time, we can easily construct arbitrarily wide registers. This can be achieved by forming an array of as many single-bit registers as needed, creating a register that holds multi-bit values (Fig. 2). The basic design parameter of such a register is its *width* – the number of bits it holds; in modern computers, registers are usually 32-bit or 64-bit wide. The contents of such registers are typically referred to as *words*.

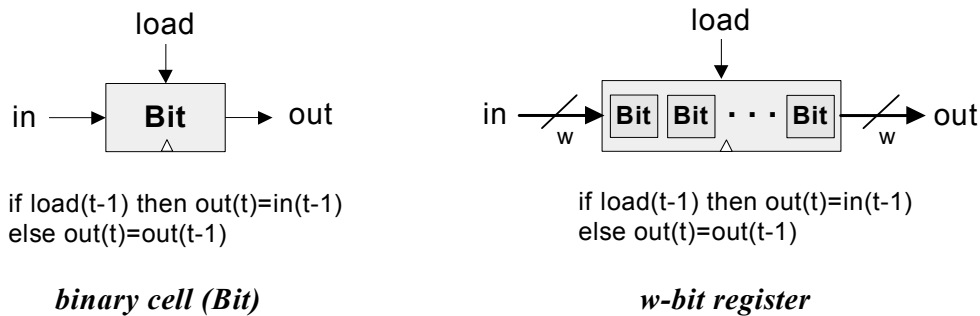


FIGURE 2: From single-bit to multi-bit registers. A multi-bit register of width w can be constructed from an array of w `Bit` chips. The operating functions of both chips is exactly the same, except that the "=" assignments are single-bit and multi-bit, respectively.

Memories: Once we have the basic ability to represent words, we can proceed to build memory banks of arbitrary length. As Fig. 3 shows, this is done by stacking together many registers to construct a *Random Access Memory* (RAM) unit. The term *random access memory* derives from the requirement that read/write operations on a RAM should be able to access randomly chosen words, with no restrictions on the order in which they are accessed. That is to say, we require that *any* word in the memory -- irrespective of its physical location -- will be accessed instantaneously, in equal speed.

This requirement can be satisfied as follows. First, we assign each word in the n -registers RAM a unique *address* (an integer between 0 to $n-1$), according to which it will be accessed. Second, in addition to stacking the n registers together, we augment the RAM chip design with a set of logic gates that, given an address j , is capable of *selecting* the individual register whose address is j .

In sum, a classical RAM device accepts three inputs: a data input, an address input, and a load bit. The *address* specifies which RAM register should be accessed in the current time unit. In the case of a read operation ($load=0$), the RAM's output immediately emits the value of the selected register. In the case of a write operation ($load=1$), the selected memory register will commit the input value in the next time unit, at which point the RAM's output will start emitting it.

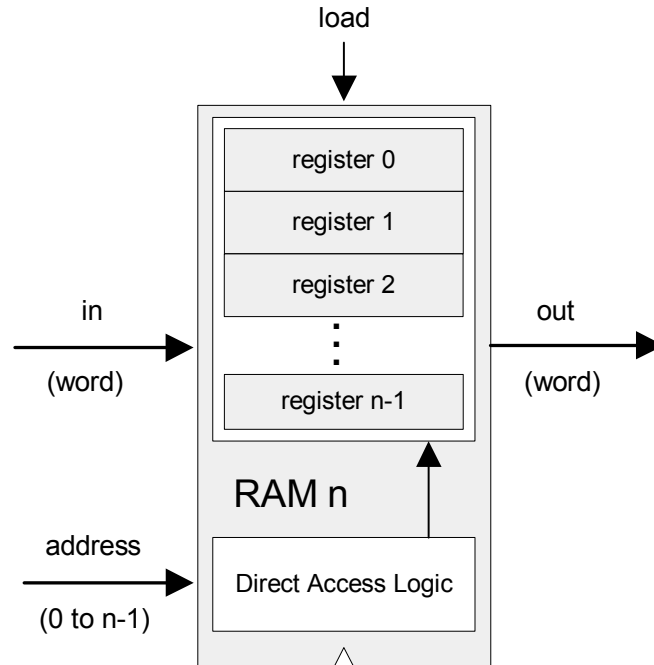


FIGURE 3: RAM chip (conceptual). The width and length of the RAM can vary.

The basic design parameters of a RAM device are its data *width* -- the width of each one of its words, and its *size* -- the number of words in the RAM. Modern computers typically employ 32- or 64-bit wide RAMs whose size is up to hundreds of millions.

Counters: A counter is a sequential chip whose state is an integer number that increments every time unit, effecting the function $out(t) = out(t-1) + c$, where c is typically 1. Counters play an important role in digital architectures. For example, most CPU's include a *program counter* that keeps track of the address of the instruction that should be executed next in the current program.

A counter chip can be implemented by combining the input/output logic of a standard register with the combinatorial logic for adding the constant 1. Typically, the counter will have to be equipped with some additional functionality, such as possibilities for resetting the count to zero, loading a new counting base, or decrementing instead of incrementing.

Time Matters

All the chips that were described above are *sequential*. Simply stated, a sequential chip is a chip that includes one or more DFF gates, either directly or indirectly. Functionally speaking, the DFF gates endow sequential chips with the ability to either maintain state (as in memory units), or to operate on state (as in counters). Technically speaking, this is done by forming feedback loops inside the sequential chip (see Fig. 4). In combinational chips, where time is neither modeled nor recognized, the introduction of feedback loops will be problematic: the output would depend on the input, which itself would depend on the output, and thus the output would depend on itself. On the other hand, there is no difficulty in feeding the output of a sequential chip back into itself, since the DFFs introduce an inherent time delay: the output at time t does not depend on itself, but

rather on the output at time $t-1$. This property guards against the uncontrolled “data races” that would occur in combinational chips with feedback loops.

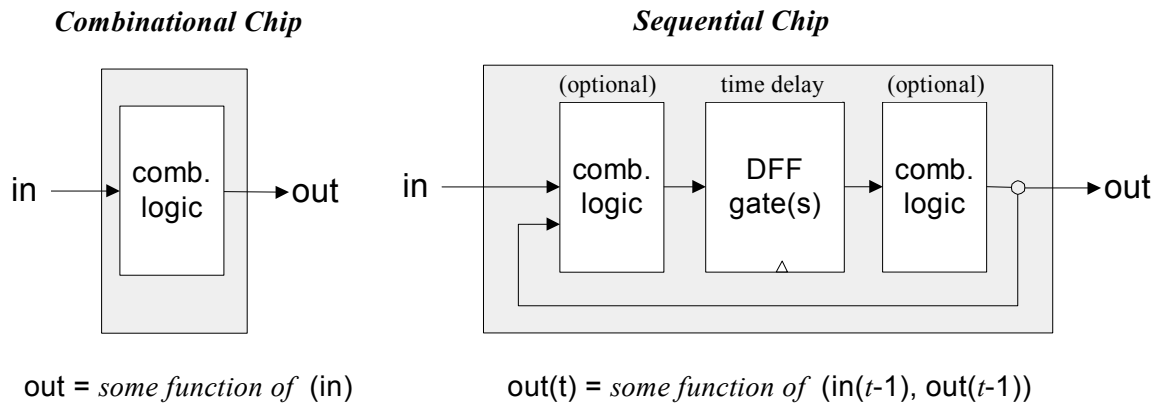


FIGURE 4: Combinational versus sequential logic (*in* and *out* stand for potentially several input and output variables). Sequential chips always consist of one layer of DFFs and optional combinational logic layers.

Recall that the outputs of combinational chips change when their inputs change, irrespective of time. In contrast, the special architecture of sequential chips implies that their outputs change only at the point of transition from one clock cycle to the next, and not within the clock cycle itself. In fact, we allow sequential chips to be in unstable states *during* clock cycles, requiring only that by the beginning of the next cycle they will output correct values.

This “discretization” of the sequential chips outputs has an important side effect: it is used to synchronize the overall computer architecture. To illustrate, suppose we instruct the arithmetic logic unit (ALU) to compute $x + y$ where x is the value of a nearby register and y is the value of a remote RAM register. Because of various physical constraints (distance, resistance, interference, random noise, etc.) the electrons representing x and y will arrive to the ALU at different times. However, being a *combinational chip*, the ALU is insensitive to the concept of time -- it continuously adds up whichever data values happen to lodge in its inputs. Thus, it will take some time before the ALU’s output will stabilize to the correct $x + y$ result. Until then, the ALU will generate garbage.

How can we overcome this difficulty? Well, since the output of the ALU is always routed to some sort of a sequential chip (a register, a RAM location, etc.), *we don’t really care*. All we have to do is ensure that the length of the clock cycle will be slightly longer than the time it takes an electron to travel the longest distance from one chip in the architecture to another. This way, we are guaranteed that by the time the sequential chip will update its state (at the beginning of the next clock cycle), the inputs that it will receive from the ALU will be correct. This, in a nutshell, is the trick that synchronizes a set of stand-alone hardware components into a well-coordinated system, as we will see in Chapter 5.

2. Specification

This section specifies a hierarchy of sequential chips:

- D-Flip-flops (DFF)
- Registers (based on DFF's)
- Memory banks (based on registers)
- Counter chips (also based on registers)

D-Flip-Flop

The most elementary storage device that we present – the basic component from which all memory elements will be designed – is the *Data Flip-Flop* gate. A DFF gate has a single-bit input and a single-bit output, as follows:



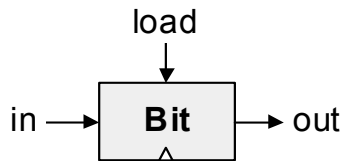
Chip name:	DFF
Inputs:	in
Outputs:	out
Function:	$out(t) = in(t-1)$
Comment:	This clocked gate has a built-in implementation and thus there is no need to implement it.

Like Nand gates, DFF gates enter our computer architecture at a very low level. Specifically, all the sequential chips in the computer (registers, memory, and counters) are based on numerous DFF gates. All these DFFs are connected to the same master clock, forming a huge distributed “chorus line”. At the beginning of each clock cycle, the outputs of *all* the DFFs in the computer commit to their inputs during the previous time-unit. At all other times, the DFFs are “latched,” meaning that changes in their inputs have no immediate effect on their outputs. This remarkable conduction feat is done in parallel, many times each second (depending on the clock frequency).

In hardware implementations, the time-dependency of the DFF gates is achieved by simultaneously feeding the master clock signal to all the DFF gates in the platform. Hardware simulators emulate the same effect in software. As far the computer architect is concerned, the end result is the same: the inclusion of a DFF gate in the design of any chip ensures that the overall chip, as well as all the chips that depend on it up the hardware hierarchy, will be “automatically” time-dependent. These chips are called *sequential*, by definition.

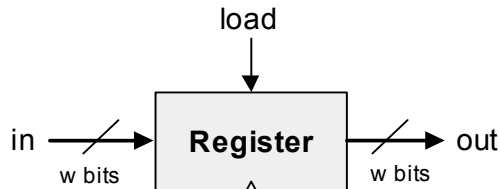
Registers

A single-bit register, which we call `Bit`, or *binary cell*, is designed to store a single bit of information (0 or 1). The chip interface consists of an input pin which carries a data bit, a `load` bit which enables the cell for writes, and an output pin which emits the current state of the cell. The interface diagram and API of a binary cell are as follows:



Chip name:	<code>Bit</code>
Inputs:	<code>in, load</code>
Outputs:	<code>out</code>
Function:	If <code>load(t-1)</code> then <code>out(t)=in(t-1)</code> else <code>out(t)=out(t-1)</code>

The API of the `Register` chip is essentially the same as that of a binary cell, except that the input and output pins are designed to handle multi-bit values:



Chip name:	<code>Register</code>
Inputs:	<code>in[16], load</code>
Outputs:	<code>out[16]</code>
Function:	If <code>load(t-1)</code> then <code>out(t)=in(t-1)</code> else <code>out(t)=out(t-1)</code>
Comment:	"=" is a 16-bit operation.

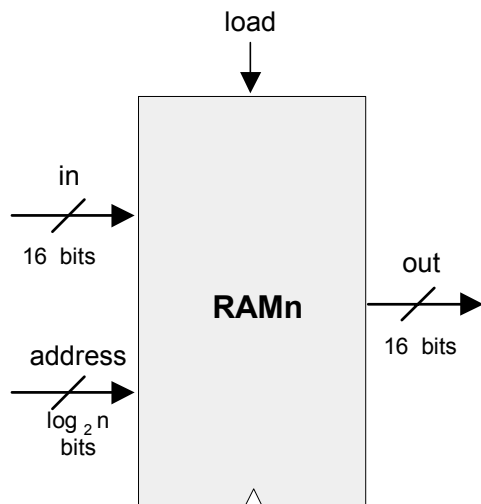
The `Bit` and `Register` chips have exactly the same read/write behavior, as follows:

Read: To read the contents of a register, we simply probe its multi-bit output.

Write: To write a new multi-bit data value d into a register, we put d in the `in` input and assert the `load` input. In the next clock cycle, the register will commit to the new data value, and its output will start emitting d .

Memory

A direct-access memory unit, also called `RAM`, is an array of n w -bit registers, equipped with direct access circuitry. The number of registers (n) and the width of each register (w) are called the memory's *size* and *width*, respectively. We will build a hierarchy of such `RAM` units, all 16-bit wide, but with varying sizes: `RAM8`, `RAM64`, `RAM512`, `RAM4K`, and `RAM16K` units. All these memory chips have precisely the same API, and thus we describe them in one parametric diagram, as follows:



Chip name: RAMn // n and k are listed below
Inputs: in[16], address[k], load
Outputs: out[16]
Function: Out (t) = RAM[address (t)] (t)
 If load (t-1) then
 RAM[address (t-1)] (t) = in (t-1)
Comment: "=" is a 16-bit operation.

We need 5 such chips, as follows:

Chip name	n	k
RAM8	8	3
RAM64	64	6
RAM512	512	9
RAM4K	4096	12
RAM16K	16384	14

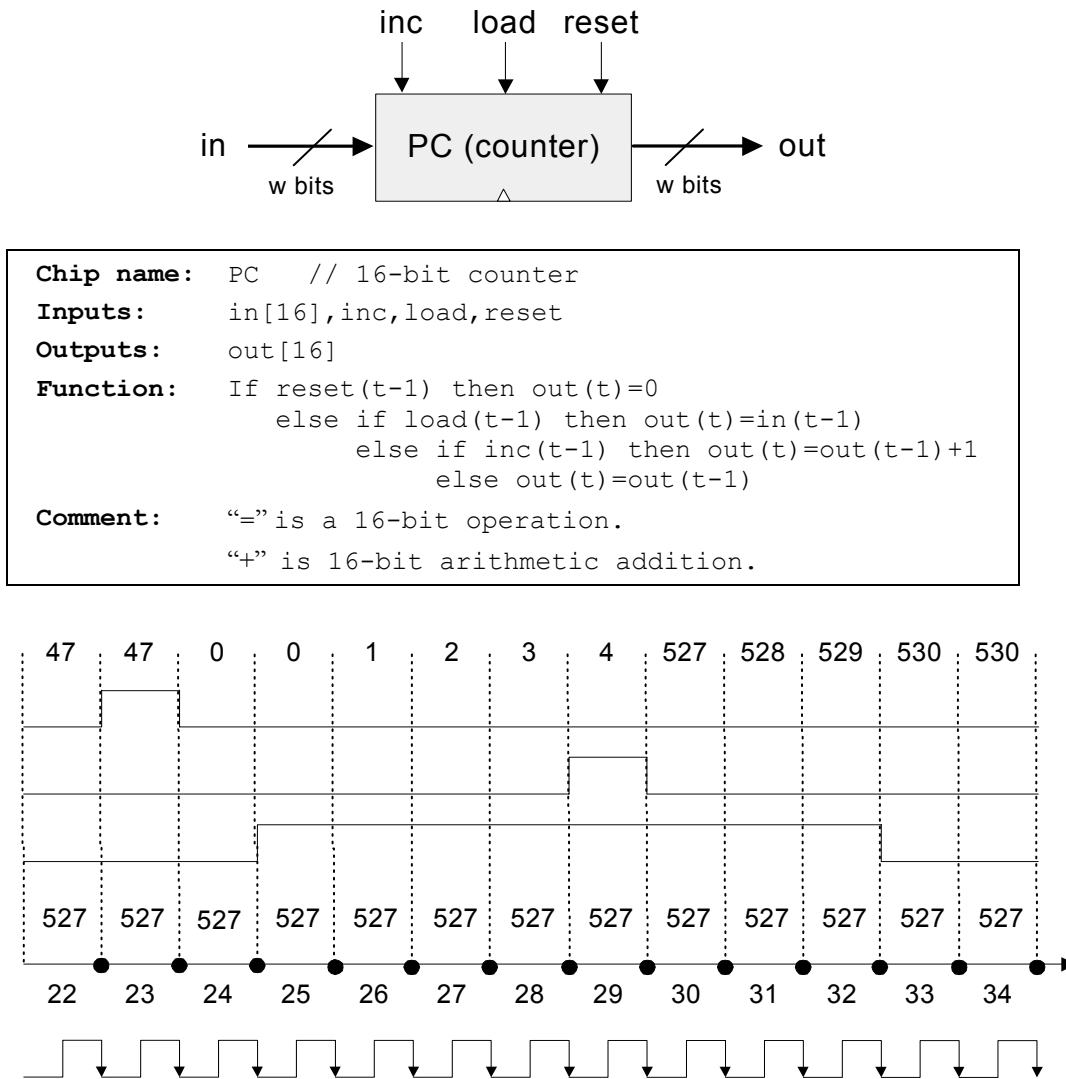
Read: To read the contents of register number m , we put m in the address input. The RAM's direct-access logic will select register number m , which will then emit its output value to the RAM's output pin. This is a combinational operation, independent of the clock.

Write: To write a new data value d into register number m , we put m in the address input, d in the in input, and assert the load input bit. The RAM's direct-access logic will select register number m , and the load bit will enable it. In the next clock cycle, the selected register will commit to the new value (d), and the RAM's output will start emitting it.

Counter

Although a *counter* is a stand-alone abstraction in its own right, it is convenient to motivate its specification by saying a few words about the context in which it is normally used. For example, consider a counter chip designed to contain the address of the instruction that the computer should fetch and execute next. In most cases, the counter has to simply increment itself by 1 in each clock cycle, thus causing the computer to fetch the next instruction in the program. In other cases, e.g. in "jump to execute instruction number n ", we want to be able to set the counter to n , and then have it continue its default counting behavior: $n+1$, $n+2$, etc. Finally, the program's execution can be restarted anytime by simply setting the counter to 0, assuming that that's the address of the program's first instruction. In short, we need a loadable and resettable counter.

With that in mind, the interface of our Counter chip is similar to that of a register, except that it has two additional control bits, labeled `reset` and `inc`. When `inc=1`, the counter increments its state in every clock cycle, emitting the value $out(t) = out(t-1) + 1$. If we want to reset the counter to 0, we assert the `reset` bit; if we want to initialize it to some other counting base d , we put d in the `IN` input and assert the `load` bit. The details are given in the counter API, and an example of its operation is depicted in Fig. 5.



We assume that we start tracking the counter in time unit 22, when its input and output happen to be 527 and 47, respectively. We also assume that the counter's control bits (`reset`, `load`, `inc`) are 0 -- all arbitrary assumptions.

FIGURE 5: Counter Simulation. At time 23 a `reset` signal is issued, causing the counter to emit zero in the following time-unit. The zero persists until an `inc` signal is issued at time 25, causing the counter to start incrementing, one time-unit later. The counting continues until at time 29 the `load` bit is asserted. Since the counter's input holds the number 527, the counter is reset to that value in the next time-unit. Since `inc` is still asserted, the counter continues incrementing, until time 33, when `inc` is de-asserted.

3. Implementation

Flip-Flop: DFF gates can be implemented from lower-level logic gates like those built in Chapter 1. However, in this book we view DFF gates as primitive, and thus they can be used in hardware construction projects without worrying about their internal implementation.

1-bit register (Bit): The implementation of this chip was given in Fig. 1.

Register: The construction of a w -bit Register chip from binary cells is straightforward. All we have to do is construct an array of w Bit gates and feed the register's load input to all of them.

8-Registers Memory (RAM8): An inspection of Fig. 3 may be useful here. To implement a RAM8 chip, we line up an array of 8 registers. Next, we have to build combinational logic that, given a certain address value, takes the RAM8's in input and loads it into the selected register. In a similar fashion, we have to build combinational logic that, given a certain address value, selects the right register and pipes its out value to the RAM8's out output. Tip: the combinational logic mentioned above was already implemented in Chapter 1.

n -Registers Memory: A memory bank of arbitrary length (a power of 2) can be built recursively from smaller memory units, all the way down to the single register level. This view is depicted in Fig. 6. Focusing on the right hand side of the figure, we note that a 64-register RAM can be built from an array of eight 8-register RAM chips. To select a particular register from the RAM64 memory, we use a 6-bit address, say $xxxyyy$. The MSB xxx bits select one of the RAM8 chips, and the LSB yyy bits select one of the registers within the selected RAM8. The RAM64 chip should be equipped with logic circuits that affect this hierarchical addressing scheme.

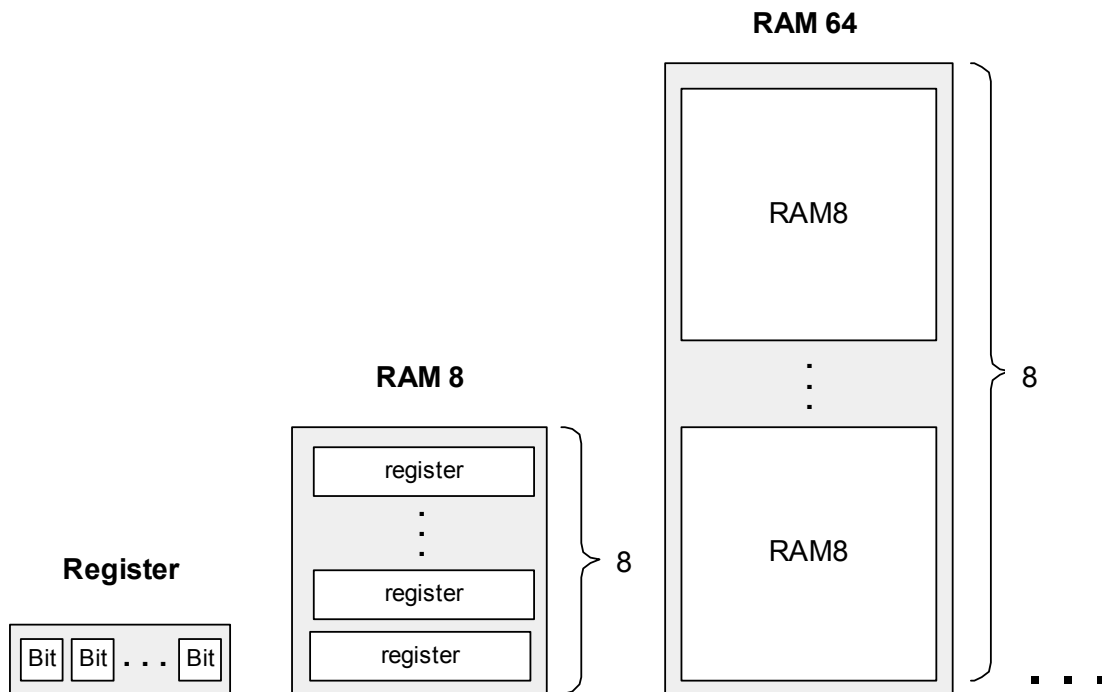


FIGURE 6: Gradual construction of memory banks by recursive ascent. A w -bit register is an array of w binary cells, an 8-register RAM an array of eight w -bit registers, a 64-register RAM an array of eight RAM8 chips, and so on. Only three more similar construction steps are necessary to build a 16K RAM chip.

Counter: A w -bit counter consists of two main elements: a regular w -bit register, and combinational logic. The combinational logic is designed to (a) compute the counting function, and (b) put the counter in the right operating mode, as mandated by the values of its three control bits. Tip: most of this logic was already built in Chapter 2.

4. Perspective

The cornerstone of all the memory systems described in this chapter is the *flip-flop* – a gate that we treated here as an atomic, primitive building block. The usual approach in hardware textbooks is to construct flip-flops from elementary combinatorial gates (e.g. Nand gates) using appropriate feedback loops. The standard construction begins by building a simple (non-clocked) flip-flop that is bi-stable, i.e. that can be set to be in one of two states. Then a clocked flip-flop is obtained by cascading two such simple flip-flops, the first being set when the clock *tics* and the second when the clock *tocks*. This “master-slave” design endows the overall flip-flop with the desired clocked synchronization functionality.

These constructions are rather elaborate, requiring an understating of delicate issues like the effect of feedback loops on combinatorial circuits, as well as the implementation of clock cycles using a two-phase binary clock signal. In this book we have chosen to abstract away these low-level considerations by treating the flip-flop as an atomic gate. Readers who wish to explore the internal structure of flip-flop gates can find detailed descriptions in [Mano, chapter 6] and [Hennessy & Patterson, appendix B].

In closing, we should mention that memory devices of modern computers are not always constructed from standard flip-flops. Instead, modern memory chips are usually very carefully optimized, exploiting the unique physical properties of the underlying storage technology. Many such alternative technologies are available today to computer designers; as usual, which technology to use is a cost-performance issue.

All the other chip constructions in this chapter -- the registers and memory chips that were built on top of the flip-flop gates -- were rather standard.

5. Build It

Objective: Build the chips listed below (except the first one). The only building blocks that you can use are primitive DFF gates, chips that you will build on top of them, and chips built in previous chapters.

▪ DFF	Data Flip-Flop (primitive – no need to implement)
▪ Bit	1-bit binary cell
▪ Register	16-bit
▪ RAM8	16-bit / 8-register memory
▪ RAM64	16-bit / 64-register memory
▪ RAM512	16-bit / 512-register memory
▪ RAM4K	16-bit / 4,096-register memory
▪ RAM16K	16-bit / 16,384-register memory
▪ PC	16-bit counter

Resources: The main tool that you will use in this project is the hardware simulator supplied with the book. All the chips should be implemented in the HDL language specified in appendix A.

As usual, for each chip mentioned above we supply a skeletal `.hdl` program with a missing implementation part, a `.tst` script file that tells the hardware simulator how to test it, and a `.cmp` “compare file.” All these files are packed in one file named `project3.zip`. Your job is to complete the missing implementation parts of all the `.hdl` programs.

Contract: When loaded into the hardware simulator, your chip design (modified `.hdl` program), tested on the supplied `.tst` file, should deliver the behavior specified in the supplied `.cmp` file. If that is not the case, the simulator will let you know.

Tip: When your HDL programs invoke chips that you may have built in previous projects, it is recommended to use the built-in versions of these chips instead. This will ensure correctness and speed up the simulator’s operation. There is a simple way to accomplish this convention: make sure that your project directory includes only the files that belong to the present project.

Likewise, when constructing RAM chips from smaller ones, we recommend to use built-in versions of the latter. Otherwise, the simulator may run very slowly or even out of (real) memory space, since large RAM chips contain many tens of thousands of lower level chips, and all these chips must be simulated as software objects by the simulator. Thus, we suggest that after you complete the implementation and testing of a RAM chip, you will move its respective HDL file out from the project directory. This way, the simulator will resort to using the built-in versions of these chips.

Steps: We recommend proceeding in the following order:

0. Before starting this project, read sections 6 and 7 of Appendix A.
1. Create a directory called `project3` on your computer;
2. Download the `project3.zip` file and extract it to your `project3` directory;
3. Build and simulate all the chips.