3. Sequential Logic [1]

Time is the substance from which I am made. Time is a river which carries me along, but I am the river; it is a tiger that devours me, but I am the tiger; it is the fire that consumes me, but I am the fire.

Jorge Luis Borges (1899-1986)

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. Combinational 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, computer architectures are 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 building blocks, we will specify and build all the memory elements 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 that we need in order to build an entire computer – a challenge that we take up in the next chapter.

Following a brief overview of sequential logic, section 3.1 introduces the notion of memory units and the sequential chips on which they are based. Sections 3.2 and 3.3 describe the chips specifications and implementation, respectively. As usual, all the chips mentioned in the chapter can be built and tested on you home computer, following the instructions given in the last section.

3.1 Background

Typical computer architectures consist of two types of chips: *combinational* and *sequential*. The outputs of combinational chips depend only on their inputs. In contrast, the outputs of sequential chips are also functions of *time*. This enables sequential chips to change their *state* as time progresses: at any given time, the state of a sequential chip is a function both of the chip's inputs and of the chip's previous state. In order to facilitate such sequential behavior, we need to represent the progression of time inside the computer.

Clocked chips: Physical time is continuous: there is no "atomic time unit". Yet dealing with changes in time becomes much easier if we can artificially make time discrete. Indeed this is what is usually done inside computers: the passage of time is marked by a master clock that tick-tacks continuously. The elapsed time between the beginning of a "tick" and the end of the subsequent "tack" is called *cycle*, and each clock cycle is treated as a discrete time unit. Unlike combinational chips, which respond to changes in their inputs instantaneously, sequential chips are made to change their outputs only at the point of transition from one clock cycle to the next, and not within the clock cycle itself. More precisely, we allow sequential chips to be in unstable

[1] From *The Digital Core*, by Nisan & Schocken, forthcoming in 2003, www.idc.ac.il/csd

states *during* clock cycles, requiring only that by the end of the cycle they will output correct values.

Sequential chips are designed to operate on *state*. For example, memory chips must maintain state, and counter chips must change it. In other words, the state of a sequential chip at time *t* should be a function of the state of the chip at time *t-1*, and of the chip inputs at time *t-1*. In order to implement such a time-dependent function, we can feed the chip output back into itself, as input. In combinational chips, such feedback loops are problematic: the output depends on the input, which itself depends 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: the discrete time-dependent behavior of the chip ensures that the output at time *t* would depend only on the inputs at time *t-1* and the output at time *t-1*. Thus the output at any given time does not depend on itself, avoiding the uncontrolled "data races" that would occur in combinational chips with feedback loops.

In sum, the clean dependence of the state at each time on the state at the previous time unit allows the safe introduction of feedback loops into the architecture. This is illustrated in Fig. 3-1, where two chips are designed to compute functions *f* and *g* using combinational circuits. The combinational chip responds immediately to any changes in the in input; in contrast, the sequential chip changes its output value only between clock cycles.

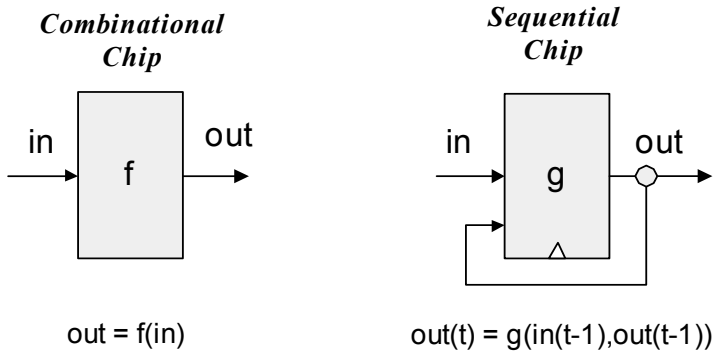


FIGURE 3-1: Combinational versus sequential logic (in this diagram in and out stand for potentially several input and output variables). The output of a combinational chip changes when its inputs change, irrespective of time. In contrast, the outputs of sequential chips change only at the beginning of the next clock cycle. The small triangle icon at the bottom of the sequential chip represents the master clock signal.

The exact implementation of clock cycles in hardware is usually done using an oscillator that alternates continuously between two phases, labeled 0 and 1. A *clock cycle* is then composed of a 0-phase followed by a 1-phase. The current clock phase (0 or 1) is a simple digital signal which is broadcast through the computer circuitry simultaneously to every sequential chip in the architecture. This way, all the sequential chips in the architecture (typically, millions of them) compute their new outputs together, at precisely the same time.

Flip-flops: The most elementary time-based sequential gate in the computer is a device whose state consists of a single bit. Such a device is called *flip-flop*, and at any given time it can be in one of two different states, labeled "flip" or "flop". There are various types of flip-flips, differing from each other in the exact way that their state is set and reset. The flip-flop that we describe in

this chapter has a single input (in) and a single output (out), and the function that it computes is out(t)=in(t-1), where t is the current clock cycle. In other words, this flip-flop simply remembers the input value from the previous time unit. Another way to describe the flip-flop's operation is to observe that it simply introduces a delay of exactly 1 time unit. As the chapter unfolds, we will show how 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.

Memory: Once we have the basic ability to remember a single bit using a flip-flop, we can easily construct arbitrarily large memory units. To get started, we can augment the flip-flop with an explicit *write-enable* signal, implemented by a single-bit *load* input (in addition to the existing in input). The resulting chip, called *binary cell* or *single-bit register*, is designed to change its stored value to the in value only when the load bit is enabled. At all other times, the chip maintains and outputs its previous state (see middle of Fig. 3-2).

By putting many single-bit registers in parallel to each other we can obtain a register that holds a multi-bit value (right of Fig. 3-2). Thus the basic design parameter of 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*.

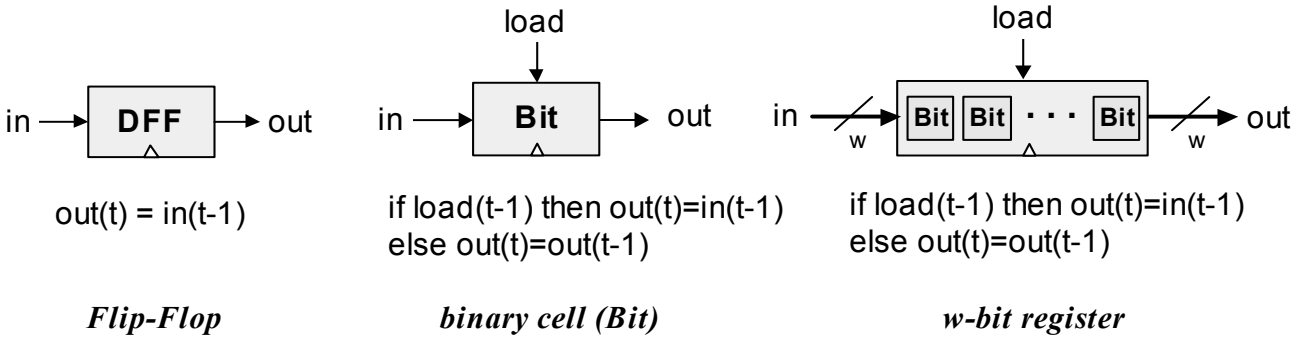


FIGURE 3-2: From flip-flop gates to multi-bit registers. A single-bit binary cell (also called Bit) is essentially a DFF gate with a loading capability. A multi-bit register of width *w* is an array of *w* Bit gates. The operating function of the register is exactly the same as that of the binary cell, except that the "=" assignments are multi-bit rather than single-bit. **Note:** Interface diagrams of sequential chips don't show their clock-regulated output-input loops, since these loops are part of the internal chip implementation.

At this point we can stack together many registers to construct a *random access memory* (RAM) – see Fig. 3-3. We need to be able to access each one of the RAM's words at will, so each word is assigned an *address* according to which it is accessed. Thus, a RAM device accepts a data input and an address input that specifies which word is accessed in the current time unit, causing the RAM to read from, or write into, the selected register.

The basic design parameters of a RAM device are its data *width* -- the width of each one of its underlying 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. The term "Random Access Memory" derives from the requirement that read/write operations on individual registers

in the RAM should be completed at the same time, irrespective of the register's location in the memory. This requirement is implemented by the RAM's direct access logic.

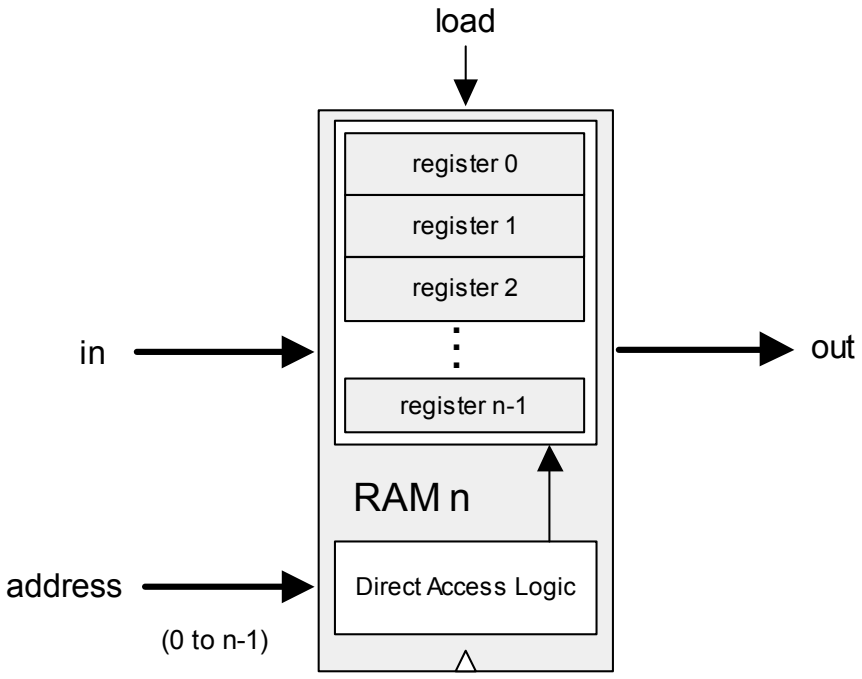


FIGURE 3-3: RAM chip (conceptual). The width of the memory has little impact on the RAM implementation, and thus 16-bit, 32-bit, and 64-bit memories have essentially the same structure. The RAM structure is scaleable in terms of *n*, meaning that its length can be easily extended.

Counters: A counter is a sequential chip that effects the function out(t)=out(t-1)+1. Counters play an important role in digital architectures. For example, the *program counter* is a control chip that keeps track of which instruction should be executed next in the presently running program. The state of a counter is an integer number that increases by one every time unit. Such a device can be implemented by combining the combinatorial logic for adding 1 with a register. In many cases some special functionality is added to a counter, such as possibilities for resetting the count to zero, loading a new value (the counting base) from the outside, or decrementing instead of incrementing. The basic design parameter of a counter is its width.

3.2 Specification

This section specifies a hierarchy of sequential gates:

- D-Flip-flop
- Registers (single-bit and multi-bit)
- Memory banks (based on registers)
- Counter chip (based on a register)

D-Flip-Flop

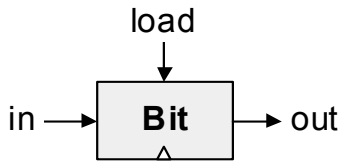
The most elementary storage device in the computer – the basic component from which all memory elements are designed – is the *Data Flip-Flop* gate. A DFF gate has a single-bit input and a single-bit output, as follows:

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 every new 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." This remarkable conduction feat is done in parallel, many times each second (depending on the clock frequency).

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:

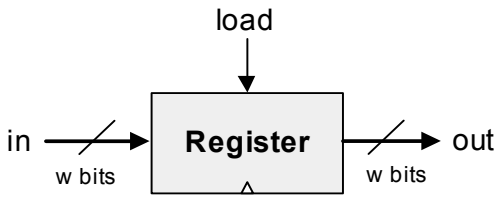


Chip name:	Bit
Inputs:	in,load
Outputs:	out
Function:	If load(t-1) then out(t)=in(t-1) else out(t)=out(t-1)

Read: To read the contents of a binary cell, we simply probe its output bit.

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

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



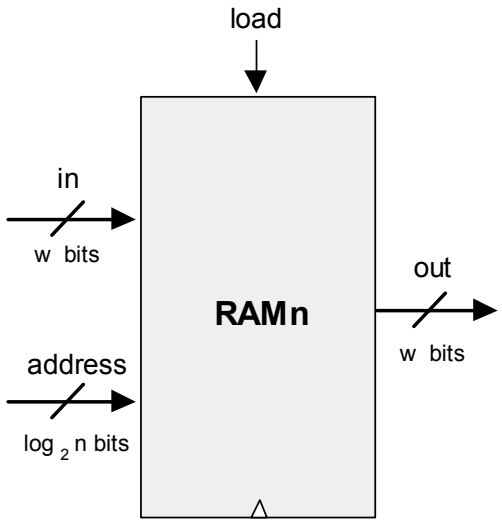
Chip name:	Register
Inputs:	in[16],load
Outputs:	out[16]
Function:	If load(t-1) then out(t)=in(t-1) else out(t)=out(t-1)
Comment:	"=" in is a 16-bit operation.

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 RAMs, all 16-bit wide, but with varying sizes.



Chip name:	RAMn // n is the RAM size
Inputs:	in[16],address[k],load
Outputs:	out[16]
Function:	Out(t)=RAM[address(t)] If load(t-1) then RAM[address(t-1)]=in(t-1)
Comment:	"=" is a 16-bit operation. We need 5 of these chips:

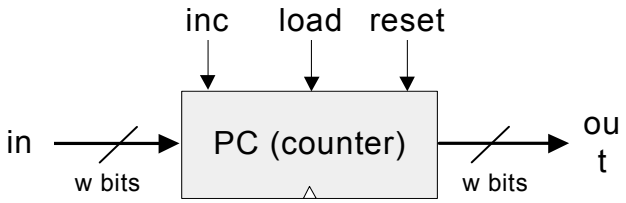
Chip name	n	k
RAM8	8	3
RAM64	64	6
RAM512	512	9
RAM4K	4096	12
RAM16K	16384	14

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 variable. 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. 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*). As a side effect, the RAM's output will emit the current value of the selected register. The new value *(d)* will become available only from the next time unit.

Counter

The counter we specify here (that will later be used as the computer's *program counter*, also called *PC*) is a loadable and resettable 16-bit counter. In other words, beyond its basic counting operation, our counter is capable of loading an external value and resetting itself to zero. The interface of the 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 or to initialize it to some other counting base, we use the reset and load control bits, respectively. The details are given in the counter API, as follows. An example of its operation is given below in Fig. 3-4.




```
Chip name:   PC   // 16-bit counter
Inputs:      in[16],inc,load,reset
Outputs:     out[16]
Function:    If reset(t-1) then out(t)=0
                else if load(t-1) then out(t)=in(t-1)
                     else if inc(t-1) then out(t)=out(t-1)+1
                          else out(t)=out(t-1)
Comment:     "=" is a 16-bit operation.
             "+" is 16-bit arithmetic addition
```

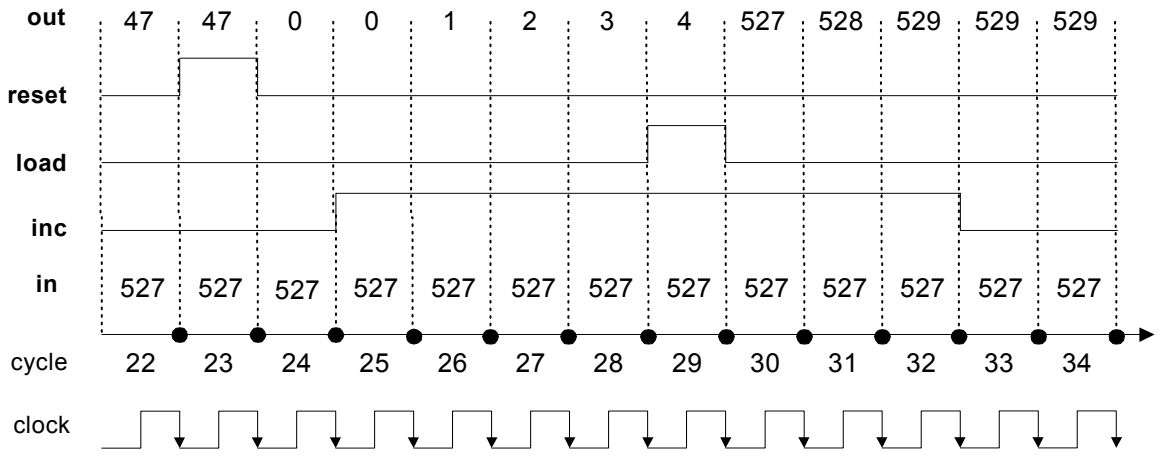


FIGURE 3-4: Counter Simulation. Suppose we start tracking the counter in time-unit 22, and that at this point the counter's out emits the value 47 and the in input holds the value 527. Finally, assume that the 3 control bits are de-asserted (all arbitrary assumptions). At time 23 a reset signal is issued, 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 starts 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.

3.3 Implementation

Flip-Flop: Although a DFF gate can be built from Nand gates, and thus need not be considered primitive, we supply a built-in DFF implementation. The reason is simulation speed. Since memory systems are based on numerous DFFs, any improvement in the basic DFF implementation can lead to dramatic performance gains throughout the computer operation. Thus we treat DFF as a primitive gate and there is no need to implement it.

Binary Cell: The main difference between a DFF gate and a Bit gate is that the latter has a load bit that enables us to reset the gate state to another value. You may obtain the functionality of the Bit gate by feeding the input of a DFF with an appropriate function of the input, load, and output values.

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 load input to all of them.

8-Registers Memory (RAM8): An inspection of Fig. 3-3 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 channels it to 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 an 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. 3-5. 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 effect this hierarchical addressing scheme.

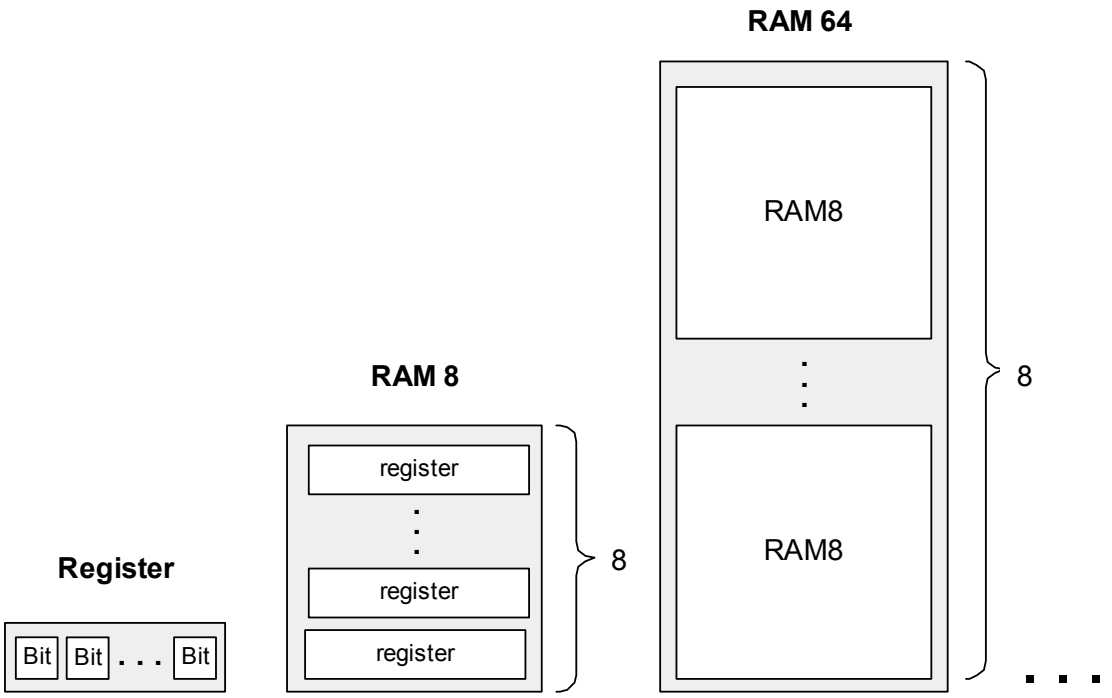


FIGURE 3-5: Gradual construction of memory banks by recursive ascent. A *w*-bit register is an array of *w* binary cells, an 8-register RAM an array of eight *w*-bit registers, a 64-register RAM an array of eight RAM8 chips, and so on. Only three more similar construction steps are necessary to build 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: the necessary combinational logic was already built in the previous chapter.

3.4 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 first uses a feedback loop to build 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 clock=1 and the second when clock=0. This "master-slave" design endows the overall flip-flop with the desired clocked synchronization functionality.

These constructions are rather elaborate, requiring understating of such delicate issues as 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 consideration by treating the flip-flop as an atomic gate. Readers who wish to drill into the internal structure of flip-flop gates can find detailed descriptions in [Mano, chapter 6] and [Hennessy & Patterson, appendix B].

The other constructions in this chapter were rather standard. At the same time, we should mention that memory devices of modern computers are usually very carefully optimized, taking into account various physical properties of the physical storage technology used to implement them. Many such alternative technologies are available today; as usual, which technology to use is a cost-performance issue.

3.5 Build It

Objective: Build the chips listed below, using primitive DFF gates and already-built chips (in this and 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

Tip: When your HDL programs invoke chips built in previous chapters, it is recommended to use the built-in versions of these chips. This will ensure correctness and speed up the simulator's operation. The built-in chips are described in detail in Appendix A.

Also, when constructing large RAM chips from smaller ones, we recommend to use built-in versions of the smaller RAM chips. Otherwise, the simulator may run very slowly or even out of 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 HDL implementation of the largest RAM chips (RAM512, RAM4K, RAM16K), you will move the respective programs `RAM512.hdl`, `RAM4K.hdl` and `RAM16K.hdl` to a separate folder on your home computer. This way, when you run the simulator on a chip that uses these chips but resides in another folder, the simulator will load the built-in versions of these chips instead of the HDL programs that you wrote.

The remaining instructions for this project are identical to those of Project 1 (section 1.5), except that every occurrence of the text "project1" should be replaced with the text "project3".