CHAPTER

The Single Cycle Datapath

So far we have been dealing with the internal design of building blocks that will form the foundations of various computer architectures. These building blocks will be aggregated in various ways to produce computer architectures with distinct cost/performance characteristics. Consider the analogy of constructing buildings. At the lowest level we have common components, various types of rooms, foundations, walls, and stairwells integrated with plumbing, wiring and temperature control devices. Each component comes in various sizes and shapes and constructed with different materials. From these basic components we can produce buildings with widely varying sizes, costs and capacities. Computer architecture can be thought of in a similar manner. By using multiplexors, ALUs, memories, and register files we can construct a datapath and interconnect them with point-to-point signals and shared buses. We refer to such an organization as a *datapath* for the obvious reason: it captures the flow of data between components that operate on and store data. We can produce architectures of widely varying costs and performance. Embedded controllers that operate in automobiles must be compact, reliable and cheap whereas supercomputers can cost millions of dollars, might be a bit more finicky, consume vastly more power but deliver the performance necessary for computationally intensive problems such as weather modeling, drug design, and crash test simulations.

This is the essence of engineering: designing and constructing systems that meet specifications for performance and reliability at a specific cost. To make appropriate trade-offs we must have an accurate understanding of the cost and performance of various implementation options. Advances in technology allow us to continually push the capability of computer architectures and thereby enable powerful supercomputers of 20 years ago to reside on our desktop at a fraction of the cost. Can you think of any other industry segment that has produced such relentless increases in performance at continually reducing cost levels?

In this chapter we begin with the idea that design of computer architectures can be viewed as the aggregation of combinational and sequential components that we have discussed to date into larger computers that you will recognize on the desktop, in your video games, and in your portable devices. However, simply aggregating components is not very meaningful unless we can orchestrate their interaction to do something useful. In this sense we start with the idea that designing an architecture is about *control*. Getting multiple large components to collectively *do* something useful can be viewed as a problem of control. We start with an elementary datapath and construct its control. This datapath is incrementally built to a modern CPU wherein we design and understand the operation of the controller (since we already understand the individual components in isolation). Then we can see how it is possible to generate a whole range of architectures by simply aggregating distinct components to satisfy a particular price-performance design point and designing the controller for such a datapath.

We start with a simple single cycle datapath.

The Single Cycle Datapath

The simplest datapath is the single cycle datapath. The basic components are a register file to store the data and functional units to operate on the data such as an adder/subtractor, logical unit, and a barrel shifter. We have constructed all of these components from basic gates and switches and should be familiar with their operation. The issue now is how can we compose larger systems with these components. How large should the data be? How many bits? In our example we will pick 32 bits, a number that is compatible with datapaths found in the majority of modern microprocessors, controllers and signal processing chips.

One of the components is a register file that has 32 registers where each register is 32-bits wide. In addition we will assume that this register file has two read ports and one write port. The block diagram is shown Figure 1 and includes the rwe (register write enable) control signal for enabling write operations to the register file. This register file can produce the values of the contents of two registers on its output ports. Addresses must be provide the two registers being read. With 32 registers we need 5 bits to specify the address of a register. Therefore we need 10 bits to specify the address of the two read registers. In addition if we are to write a register with a value we need another 5 bits to identify the register that is to receive this value and 32 bits for the value that is to be written into this register. Register 0 is special and will always contain the value 0. In the examples at the conclusion of the chapter it will become evident that this is a useful feature.

Additional functional units are shown in Figure 1. The control signals for the functional units include signals for the adder/subtractor, shift unit, and the logic unit. The adder/subtractor requires one bit to specify the operation and one bit to enable it to drive the output bus. The shifter is simi-

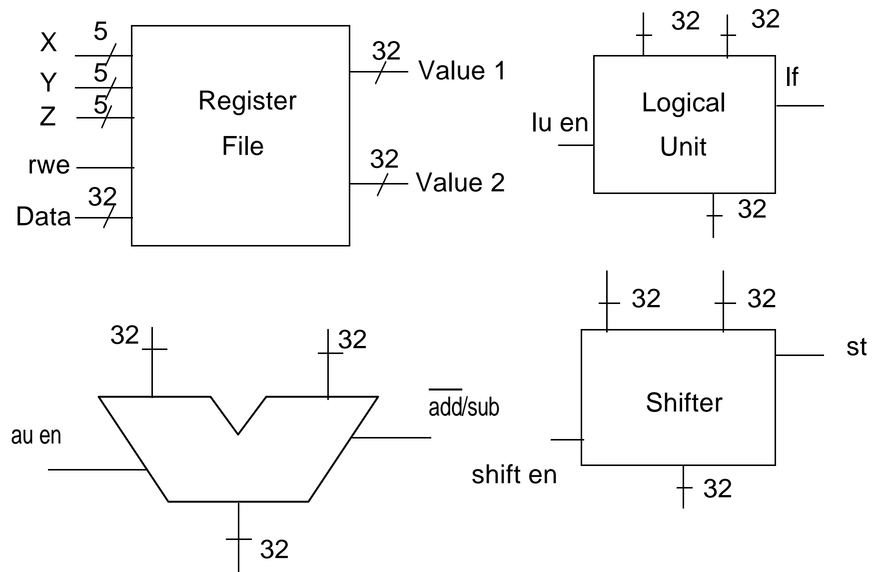

FIGURE 1. **Single cycle datapath components**

larly constructed: one input provides the data to be shifted and a second input provides the amount by which the data is to be shifted as a two's complement number. Positive numbers refer to a shift right by that amount and negative numbers refer to a shift left by that amount. In addition, a two-bit shift type field specifies whether this is a logical shift (00), arithmetic shift(01), or a rotate (10) operation. Finally, the logical unit requires four bits as input to specify any boolean function on two bits. For example, to perform the exclusive-OR operation the input pattern would be 0110. This function is applied to each pair of bits of the input operands.

These basic architectural components are aggregated into a datapath as shown in Figure 2. Each functional unit has two inputs that are provided by the register file. Each output of the register file is connected to a bus. These two buses, X and Y, serve to carry input values to the functional units. The outputs of the functional units drive a third bus, the Z bus, that is used to carry the data value to be written to the register file. We will see that it is also very useful to be able to directly specify values used in computations, for example the number 1 used in increment operations or the value 4 which is the number of bytes in a word. To facilitate the use of such constants the datapath is augmented with a special register that contains a value that can be optionally selected as one of the values on the Y bus rather than the output of the register file. We will refer to this value as an *immediate* value to distinguish it from a value that must be fetched from a register. An associated control signal, imm en, is used to determine

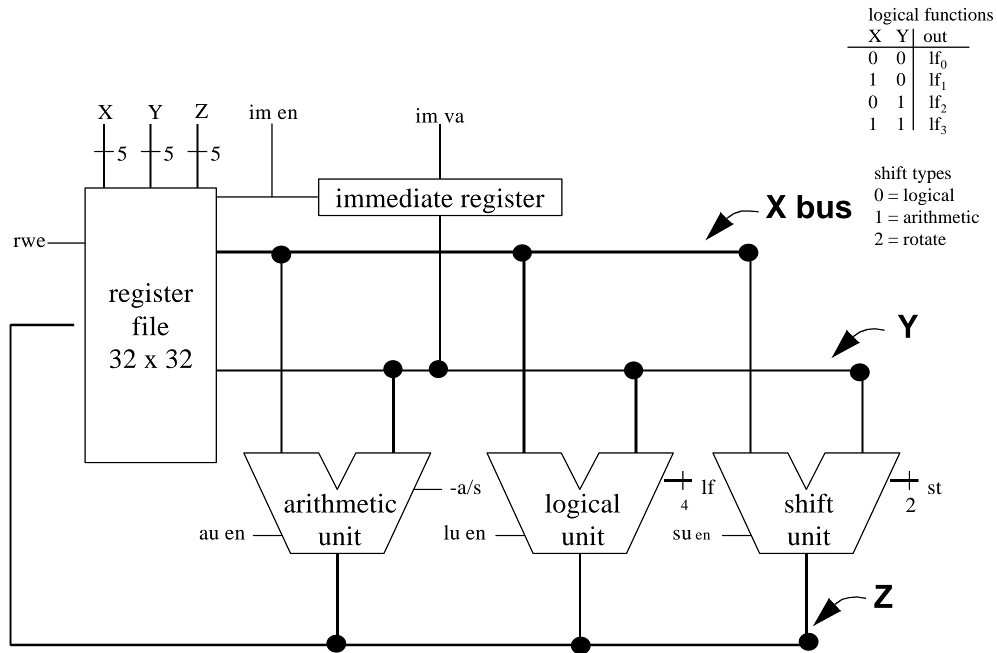

FIGURE 2. The basic single cycle datapath

the source of data on the Y bus. This signal is also used to disable the Y output of the register file when the immediate value is used. Note that the use of the *imm en* to disable the output of the register file has the same effect as placing a 2:1 multiplexor at the input of each of the functional units.

Datapath Control

Now we have connected the components in a way that values or data can flow from where they are stored in the register file, to the component that operates on them, and back to the register file. We have seen how these individual components operate and by specifying the values of all of the control signals we can control the datapath to effect certain sequence of operations. For example, consider the addition of the contents of registers R1 and R2 with the result written into R3. We can write this operation as

$$R3 = R\ 1 + R2$$

To effect this operation we need to specify the values of all of the control signals. For the preceding operation the control signal values should be set as follows.

Step	X	Y	Z	rwe	imm en	im va	au en	$a/\bar{s}$	lu en	lf	su en	st
1	1	2	3	1	0	X	1	0	0	XXXX	0	XX

The selection of 1 and 2 as the values on the X and Y ports of the register file will cause the contents of these registers to be placed on the X and Y buses. By setting the value of imm en to 0 we prevent the contents of the immediate register from being placed on the Y bus in lieu of the contents of register 2. The adder/subtractor is enabled to add the two inputs and place the result on the Z bus. The register file rwe enables the store of the value on the Z bus to the register address on the Z address port which is 3. Note that while the register addresses are identified as 1, 2, and 3 these are actually 5-bit quantities and the values on the ports of the register file are actually 00001, 00010, and 00011 respectively. The values of all of the other control signals are shown in binary form. Finally the Xs represent don't care values. For example, if the shift unit is not active (sh en = 0) then the value of the shift type (st) is inconsequential. The operation is now complete.

Often the time to perform an operation such as this is referred to as a *cycle*. This type of operation is also referred to as a *register-to-register* operation. We can create similar steps to perform other operations on register contents using the other functional units. It is also apparent that we could construct such a datapath with other basic components such as a multiplier and control it in a similar fashion.

Example: This is an example of the following logical operation: R3 = R1 XOR R2.

Step	X	Y	Z	rwe	imm en	im va	au en	$a/\bar{s}$	lu en	lf	su en	st
1	1	2	3	1	0	X	0	X	1	0110	0	XX

Example: The shift unit is useful in performing multiplications by powers of 2. For example, we might perform the following operation: R5 = 4(R6). The immediate register is used to provide the shift amount (imm va = -2). Remember that a left shift of two bits is equivalent to a multiplication by 4 and the sign of the shift value indicates the direction of the shift. The shift type is arithmetic (st = 1).

Step	X	Y	Z	rwe	imm en	im va	au en	$a/\bar{s}$	lu en	lf	su en	st
1	6	X	5	1	1	-2	0	X	0	XXXX	1	1

A few points about terminology. The set of control signals organized as sequence of bits as shown above is often referred to as a *control word*. How large is the control word for the datapath in Figure 2? Each register address requires 5 bits since there are 32 registers. The three register addresses require 15 bits. The lf field requires four bits and the shift type field, st, requires 2 bits. All other control signals are single bits leading to a total of 27 bits. We do not include the immediate value in the actual control word for reasons that will become clearer once the datapath is developed a bit more. A control word is referred to as a *microinstruction*. Associated groups of control signals are referred to as fields. For example the bits used to control the logic unit (the lf and lu en bits) may collectively be referred to as the logic unit *field*. This organization of control bits is referred to as a format. The preceding table is the format of the microinstruction for the datapath of Figure 2.

Now that we have a sense for operations that can be realized in a single step what about more complex operations? For example, suppose we wished to perform the following computation.

$$R0 = \frac{R1 + R3}{2} + 4(R2 - R0)$$

This computation breaks down into a sequence of arithmetic operations. Note that multiplication or division by a power of two is equivalent to a shift by some multiple of bits. As described earlier each arithmetic operation can be performed by appropriately setting all of the control bits. A sequence of control words or microinstructions will effect the computation of complex expressions. You can imagine this sequence of control words as a simple program except that this program is written in binary rather than some more easily decipherable (by us) language. Moreover, you are directly controlling the hardware to effect the operations rather than writing in a high level language. In fact we will refer to such a sequence of control words as a *microprogram* and such programs will be referred to as *microcode*. Thus our strategy for implementing complex expressions on this datapath is to break down the expression to a sequence of primitive operations and to implement each primitive operation, such as addition, in a single control word. By sequencing through these control words the complex expression can be evaluated. One issue may now be apparent. What about the control logic to sequence through these control words? We will get to design of the controller last.

Adding Memory to the Datapath

The register file can only hold 32, 32-bit values. We certainly write programs that utilize more values this! As a result storage is necessary for the data and memory is utilized for this purpose. The

use of memory now potentially adds an extra step to a computation. If the data we wish to process is not available in a register, but is stored at some location in memory, the data must first be moved into a register for processing. Thus a memory module must be added to the components that comprise the data path. Functionally the datapath must provide addresses to memory as well as the capture data that is returned by memory. When data is being stored in memory both an address and data must be supplied to the memory module.

As we have seen memories have a very simple functional behavior. If an address is provided, the memory can return the value of the binary number stored at that address. Alternatively if an address and a value are provided, the value can be stored in memory at that location. Control of memory behavior can be achieved with two control signals: $r/\overline{w}$ and msel. The latter enables the memory module for operation in which case the former determines if the operation being performed is a read or a write. The datapath extended to incorporate a memory module is shown in Figure 3. During a memory read cycle the following actions take place. The register that is placed on the X bus provides a 32 bit address to memory. The msel and $r/\overline{w}$ signal are asserted. After some delay the contents of memory at that address is placed by the memory module on the Z bus and is written into the destination register. During a write or store operation while the memory address is on the X bus, the contents of a second register is placed on the Y bus. This value is written to memory at the location specified by the address on the X bus. The msel signal is asserted and the $r/\overline{w}$ signal is 0 (write control is active low). The st en and ld en signals control the flow of data to or from the memory module respectively.

This modified datapath is shown in Figure 3. To load a value from some location in memory we might have the following sequence of operations.

R28 = R28 + Immediate

R3 = M[R28]

The first operation adds the contents of register R28 to the value stored in the immediate register and stores the result back into register R28. This value is used in the next step as an address. The second operation denotes the retrieval of a value stored in memory. The address from which the value is retrieved is the value stored in register R28. Similarly a store operation would appear as follows.

M[R28] = R3

To perform memory operations a few additional control signals must be given appropriate values. These are ld en, st en, r/w, and msel. The first two control the flow of data to or from memory. The latter two control memory behavior, that is, read/write operation and enable/disable functions respectively.

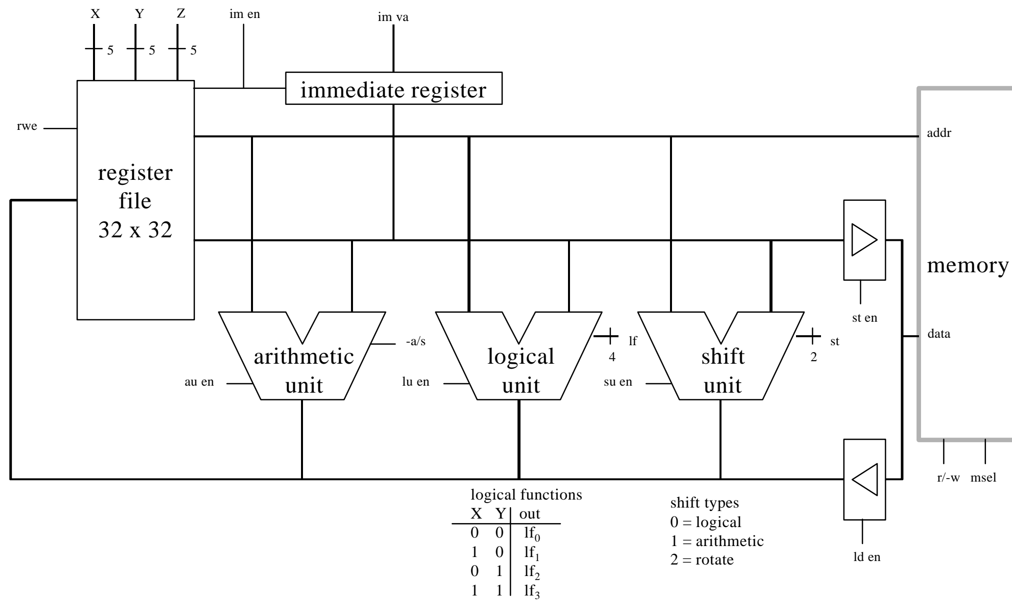

FIGURE 3. Single cycle datapath with memory and immediate operand

The control word now has four additional fields and appears as follows.

X	Y	Z	rwe	imm en	imm va	au en	$\bar{a}$/s	lu en	lf	su en	st	st en	ld en	r/$\bar{w}$	msel

TABLE 1. Control word for single cycle datapath with immediate and memory

Example: Consider a load operation R1 = M[R2]. The control word would be appear as follows.

X	Y	Z	rwe	imm en	imm va	au en	$\bar{a}$/s	lu en	lf	su en	st	st en	ld en	r/$\bar{w}$	msel
2	X	1	1	0	X	0	X	0	X	0	X	0	1	1	1

Example: Consider the store operation M[R2] = R1

X	Y	Z	rwe	imm en	imm va	au en	$\overline{a}/s$	lu en	lf	su en	st	st en	ld en	$\overline{r}/w$	msel
2	1	X	0	0	X	0	X	0	X	0	X	1	0	0	1

Addressing Modes

An *addressing mode* is the manner in which memory addresses are constructed. The datapath shown in Figure 3 has an addressing mode where the contents of a register is used as an address. This is often referred to as *register indirect* mode. However we find that there are many common ways in which addresses are computed and modern microprocessors will support other forms of addressing or addressing modes in hardware.

For example, imagine we had just finished performing the operation R1 = M[R2] and now we wish to obtain the value at the next word in memory. We can obtain the address of this word by adding 4 to the contents of R2. This is because memory is byte addressed and we have 32 bit words (four bytes/word). If we wish to access a sequence of words the address of each successive word can be obtained simply by adding 4 to the address of the previous word. We observe that for such structured accesses the address of the kth word in an array can be given by

$$\text{Address of } k^{th} \text{ word} = B + 4*k$$

where B is the address of the first word and we start counting from 0.

For the datapath of Figure 3 we would iteratively use the adder/subtractor to increment the address in a register between accesses to memory. Some modern CPUs will support this address computation in hardware. In any case this addressing mode goes by several names including *indexed addressing* and *base-offset addressing*. As the datapath evolves we will encounter other addressing modes that will be supported in hardware. As we provide more examples, it will be clear how other addressing modes can be supported in control word sequences.

General Memory Access Issues

We have focused on the access and storage of 32 bit quantities from a byte addressed memory. We should recognize that it is certainly feasible, and often desirable, to access memory in quantities of other

than 32 bits. For example, in processing image data we might wish access and process data in 8 bits quantities while for audio data we may wish to process data in 16 bit units. Modern CPU datapaths will often provide implementations that support such accesses. However, we will restrict our discussion to the access of 32 bit quantities from memory.

Examples

Several relatively more detailed examples will clarify some of the operational issues for the datapath of Figure 3.

Example 1: Complex Expression Evaluation - R3 = 3(R1- R2) + (R4/4) (Note: lsf = logical shift, asf = arithmetic shift). The sign of the shift amount determines whether the shift operation moves left or right.

X	Y	Z	rwe	imm en	imm va	au en	$\overline{a}/s$	lu en	lf	su en	st	st en	ld en	$r/\overline{w}$	msel	Descr
1	2	1	1	0	X	1	1	0	X	0	XX	0	0	X	0	R1 = R1-R2
1	X	5	1	1	-1	0	X	0	X	1	01	0	0	X	0	R5 = R1 asf 1
1	5	1	1	0	X	1	0	0	X	0	XX	0	0	X	0	R1 = R5 +R1
4	X	4	1	1	2	0	X	0	X	1	1	0	0	X	0	R4 = R4 asf 2
1	4	3	1	0	X	1	0	0	X	0	X	0	0	X	0	R3 = R1 +R4

Example: Use of the logic unit: R6 = (R2 and R4) EXOR R5

X	Y	Z	rwe	imm en	imm va	au en	$\overline{a}/s$	lu en	lf	su en	st	st en	ld en	$r/\overline{w}$	msel	Descr
2	4	2	1	0	X	0	X	1	1000	0	XX	0	0	X	0	R2 = R2 and R4
2	5	6	1	0	X	0	X	1	0110	0	XX	0	0	X	0	R6 = R2 xor R5

Example: Memory access - M[R2] = M[R4] + M[R6] + R1

X	Y	Z	rwe	imm en	imm va	au en	$\overline{a}$/s	lu en	lf	su en	st	st en	ld en	r/$\overline{w}$	msel	Descr
4	X	3	1	0	X	0	X	0	X	0	XX	0	1	1	1	R3 = M[R4]
6	X	5	1	0	X	0	X	0	X	0	XX	0	1	1	1	R5 = M[R6]
3	5	3	1	0	X	1	0	0	X	0	XX	0	0	X	0	R3 = R3 + R5
3	1	3	1	0	X	1	0	0	X	0	XX	0	0	X	0	R3 = R3 + R1
2	3	X	0	0	X	0	X	0	X	0	X	1	0	0	1	M[R2] = R3

Example: The following control word loads a value into a register: R1 = 1024. The value 1024 is provided in the immediate register. From the datapath it is clear that the path from the immediate register to register R1 is through a functional unit. We can choose to add R0 to the immediate register and store the result in R1. Alternatively we can choose to pass the immediate value through the logic unit. By setting the value of lf = 1100 the operation performed by the logic unit effectively passes the value on the Y bus through the Z bus. Look at the truth table for a two variable, X and Y, boolean function. With lf = 1100 the output column and the column for the Y input are identical. The use of the hardwired 0 value in R0 is useful here.

X	Y	Z	rwe	imm en	imm va	au en	$\overline{a}$/s	lu en	lf	su en	st	st en	ld en	r/$\overline{w}$	msel	Descr
0	X	1	1	1	1024	0	X	1	1100	0	XX	0	0	X	O	R1 = 1024

Example: Pixel averaging: To store image data 8-bits is often sufficient to for each pixel value. This we may find four pixels stored in each word. Consider the need to average the values of the four pixels in a word. This is an operation one might find in low level image processing operations. The implementation shown below successively extracts the 8 bits from a pixel using shift and logical AND operations. These 8-bit values are added in register R3. The last instruction performs a division by 4.

X	Y	Z	rwe	imm en	imm va	au en	$\overline{a}$/s	lu en	lf	su en	st	st en	ld en	r/$\overline{w}$	msel	Descr
0	X	1	1	1	1024	0	X	1	1100	0	XX	0	0	X	O	R1 = 1024
1	X	2	1	0	X	0	X	0	X	0	XX	0	1	1	1	R2 = M[R1]
2	X	3	1	1	0xFF	0	X	1	1000	0	XX	0	0	X	0	R3 = R2 and 0xFF
2	X	2	1	1	8	0	X	0	XXXX	1	00	0	0	X	0	R2 = R2 lsf 8

X	Y	Z	rwe	imm en	imm va	au en	$\overline{a}$/s	lu en	lf	su en	st	st en	ld en	r/$\overline{w}$	msel	Descr
2	X	5	1	1	0xFF	0	X	1	1000	0	XX	0	0	X	0	R5 = R2 and 0xFF
3	5	3	1	0	X	1	0	0	XXXX	0	XX	0	0	X	0	R3 = R3 + R5
2	X	2	1	1	8	0	X	0	XXXX	1	00	0	0	X	0	R2 = R2 lsf 8
2	X	5	1	1	0xFF	0	X	1	1000	0	XX	0	0	X	0	R5 = R2 and 0xFF
3	5	3	1	0	X	1	0	0	XXXX	0	XX	0	0	X	0	R3 = R3 + R5
2	X	2	1	1	8	0	X	0	XXXX	1	00	0	0	X	0	R2 = R2 lsf 8
3	2	3	1	0	X	1	0	0	XXXX	0	XX	0	0	X	0	R3 = R3 + R2
3	X	3	1	1	2	0	X	0	XXXX	1	01	0	0	X	0	R3 = R3 asf 2

Alternative Datapath Organizations

We can easily envision many alternative datapath organizations that may make use of other components such as multipliers and divide modules or even additional register files. These components may have different organization of buses that carry data between them. However, if we are familiar with the operation of each component and are provided with the control signals for each component it is possible to orchestrate computations among the modules just as we have done for the datapath of Figure 3. The challenge is breaking the required computations into a sequence of register transfer level operations each of which can be performed in a single step. Thereafter determining of control signal values is largely straightforward. The workbook problems have several examples of alternative datapath organizations.