Instruction Set Architecture

This chapter moves the level of abstraction at which the datapath is programmed up above the microcode level to the *assembly language* level. Microinstructions are replaced by assembly instructions that specify operations and data rather than the value of every control signal. As a result the datapath must be augmented with control logic that generates the control signals that implement each assembly instruction. This chapter describes the set of instructions and the modifications to the single cycle datapath to decode these instructions. It is now possible to describe how programs written in C or Java are implemented - by translating them into programs comprised of the assembly instructions.

Raising the Level of Abstraction

To this point we have written microcode by taking a computation, breaking it up into a sequence of elementary steps such that each step can be executed in one cycle on this datapath. Our description of the behavior of each step can be expressed in a *register transfer level* (RTL) notation of the following form.

$$R1 = R2 + R3$$

The notation specifies that the contents of registers R2 and R3 are to be added and the result placed in register R1. We can see that such a level of description is clearly preferable to specifying the values of

30 control signals that would implement this step in the datapath. It is compact, easier to remember, can be written with less knowledge of the hardware and is generally less error prone. This is the level at which we would prefer to describe the operation of the datapath. Describing computations at this level rather than microcode also serves as an implementation independent description. Automated tools could then translate such RTL descriptions into microcode for a specific datapath.

While the preceding expression is an easy way to write RTL expressions machine readable forms generally look quite different. The syntax used for the assembly language of the MIPS R3000/4000 series of processors is the following.

```
add $10, $8, $9
```

The 32 registers in the register file are denoted as $0, $1, $2,...,$31. This above construct is referred to as an *instruction*. Additional information is necessary to determine which of the registers correspond to sources of data and which register will be the destination of the operation. A typical convention followed in the MIPS R3000/4000 processors is the following: $8 and $9 are the sources of data or *source operands.* Register $10 is the destination of the result or *destination operand* and add is the *operation*. This instruction signifies that the contents of $8 should be added to the contents of $9 and the result should be placed in $10. From the operation of the datapath we know that such an instruction can be implemented in one control word. We can similarly introduce instructions for other arithmetic and logical operations. These additional instructions will also have source operands and destination operands in registers.

How about accesses to memory? There are two basic operations with respect to memory. The access of data from memory and the storage of data to memory. The former is referred to as memory load operation and can be expressed with the following instruction.

```
lw $10, $8
```

Register $8 contains the address of a location in memory. This address is treated as a word address, The 32-bit value stored at that address is loaded into register $10. Storage of data to memory is achieved with the memory store operation and can be expressed with the following instruction.

```
sw $10, $8
```

Register $8contains the address of a location in memory. This address is treated as a word address, The 32-bit value in register $10 stored at that address.

We now have the beginnings of a set of instructions or instruction set, namely that of the MIPS/R3000/4000.

The Instruction Set

Some of the instructions supported in the single cycle GT SPIM datapath are shown in Table 1. We will refer to these instructions as *native* instructions since each instruction can be implemented with a single microinstruction. In fact the datapath can implement 21 native operations: addition, subtraction, shift logical, shift arithmetic, rotate, and 16 logical operations. Each of these operations can be performed with a pair of source registers and a single destination register resulting in 21 native instructions. To this set we can add the lw and sw memory instructions raising the total number of native instructions to 23. Note that Table 1 only shows three logic instructions and the logic unit will support thirteen more possible combinations for which corresponding assembly language mnemonics and instructions could be defined.

While the basic operations in the datapath are fixed, flexibility in choice of operands increases the number of native instructions we might define. For example, an addi $10, $8, 4 instruction can be added where one of the source operands is the contents of the immediate value register. Other such instructions such as subi, xori and roti can also be defined. These instructions can also be executed in a single microinstruction. In principle for each of the 23 native instructions we can define another native instruction where one of the operands is drawn from the immediate value register. Of course the utility of having all of these additional instructions is another matter! Note that Table 1 does not show these additional instructions with an immediate operand.

Instruction	**Interpretation**
add $10, $8, $9	$10 = $8 + $9
sub $10, $8, $9	$10 = $8 - $9
and $10, $8, $9	$10 = $8 and $9
or $10, $8, $9	$10 = $8 or $9
xor $10, $8, $9	$10 = $8 xor $9
sa $10, $8, $9 (shift arithmetic)	The contents of $8 are shifted by the amount in $9 and placed in $10
sl $10, $8, $9 (logical)	The contents of $8 are rotated by the amount in $9 and placed in $10
rot $10, $8, $9 (rotate)	The contents of $8 are shifted by the amount in $9 and placed in $10
lw $10, $8	$10 = M[$8]
sw $10, $8	M[$8] = $10

TABLE 1. **Native Instructions supported by the GT SPIM datapath**

The Big Picture

So where do these assembly instructions exist in our everyday experience of writing and executing C or Java programs? This is best captured in Figure 1. High level language programs written in, say C or Java, are first translated into sequences of assembly language instructions. This process is referred to as compilation and is performed by a *compiler*. In this process variables in the source program become registers or memory locations in the assembly program. Note that we manipulate variables and abstract data structures in a source program. In contrast assembly instructions manipulate hardware entities such as registers, adder/subtractors, and memories. Thus, we see that the compiler is burdened with the task of taking variables and data structure elements from the source program and placing them in registers or memory and generating the instructions that perform the operations described in the source program. Each of these assembly instructions may now be decoded to produce the control signals that cause that assembly instruction to be executed. The design of this decoding logic will be the last topic that we address in this chapter after we have completed a discussion of a few more elements of this datapath. In particular, before we can begin decoding instructions we must understand how these assembly instructions are encoded in the first place.

Instruction Formats

Several different types of instructions were defined in the previous section. From Figure 1 it is apparent that these instructions must eventually be translated into the microinstructions that drive the datapath. The decode logic that performs this translation operates on an instruction to produce the correct values of the control signals. How are these instructions stored? Like everything else in digital systems, an instruction is stored as a binary pattern. Since the GT SPIM datapath operates on 32-bit quantities there is an inherent advantage in encoding each instruction into 32-bit words. The specific rules for encoding instructions into 32-bit words is captured in the instruction format. At this point we can distinguish two basic instruction formats: register format and immediate format.

Register Format

Just as control words have fields and formats, instructions are encoded in binary according to a format fixed by the machine designers. Consider the following register-to-register instruction.

```
add $10, $8, $9
```

An encoding into a 32-bit binary number must record the operation (addition) the source operands (registers $8 and $9) and the destination operand (register $10). The R-format used to encode

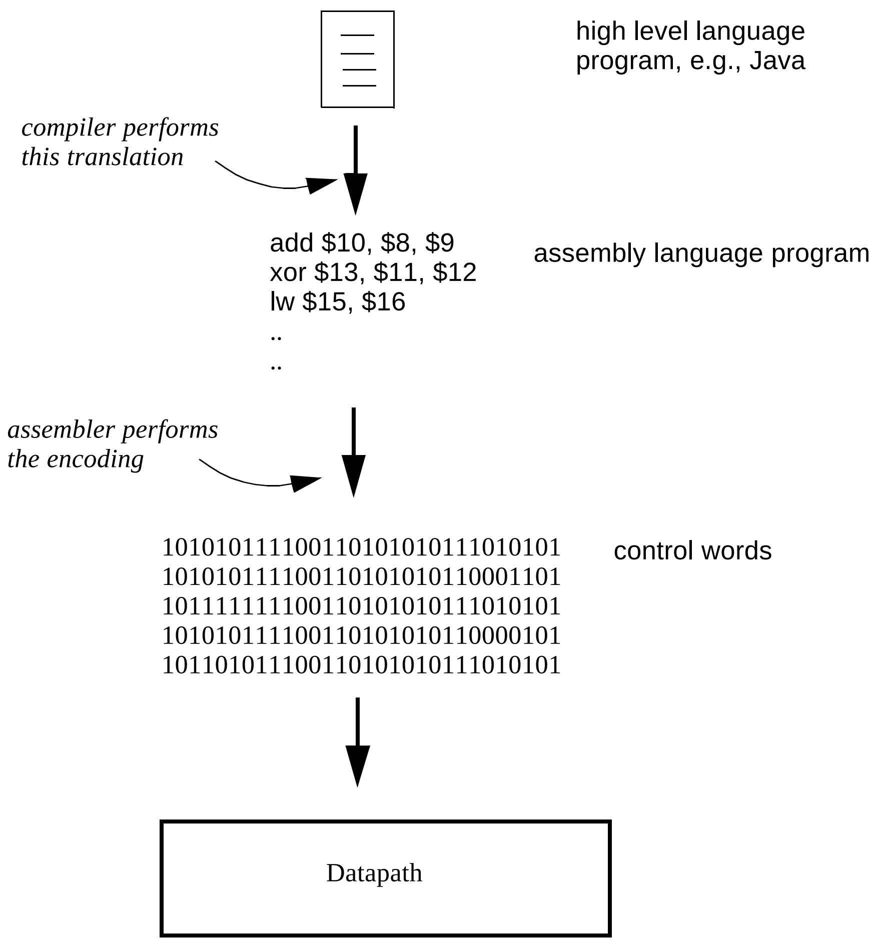

FIGURE 1. **Implementing high level language programs**

such register-to-register instructions is shown in Figure 2. The *opcode* is a 6-bit field that encodes the type of operation, for example, an addition operation. A 6-bit opcode enables us to define up to 64 instructions. The source and destination operands are specified using 5-bit fields since each field can specify up to 32 registers. The remaining bits are currently unused but will find application later in the development of the datapath. Let us assume that the opcode for register-to-register addition is 100100, and that the unused bits have a value of 0. Registers $10, $8, and $9 are refer to registers numbered 10, 8, and 9. With 32 registers in the datapath 5 bits are required to encode each of the registers used in the instruction. The resulting encoding of the above instruction is 0x91484800. All register-to-register operations are encoded in this manner producing a 32-bit binary encoding.

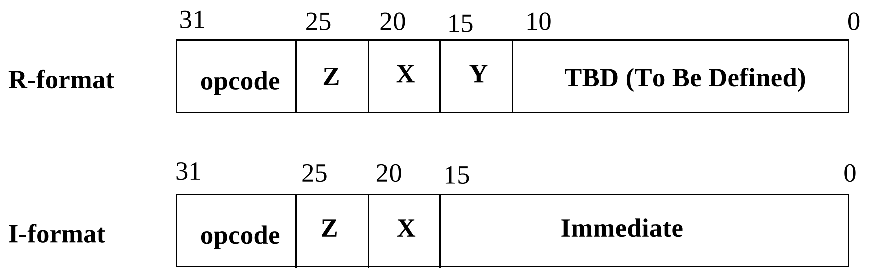

FIGURE 2. **Instruction formats**

A second class of instructions are the memory operations. The load word instruction, lw, requires two registers. For example, an instruction that loads a register with a word from memory is written as

lw $10, $28

In this case the register $10 is loaded with the contents of memory at the address given by the contents of register $28. The load instruction can be encoded using the R-format. The register containing the address is encoded in the X field while the destination register is encoded in the Z field. The bits in the remaining fields are set to 0.

The store instruction is structured similarly and can also be encoded using the same format.

sw $10, $28

The contents of $10 are stored in memory at the address given by the contents of $28.

Immediate Format

Often we encounter operations where one of the operands is a constant. For example, it is common to increment a value by 1 or to multiply a number by 2. In these cases it would be convenient to define an instruction format wherein one of the operands is a constant rather than a register. One such format is the *I-format* shown in Figure 2. To understand how this format is used consider the following instruction.

```
addi $10, $8, 4
```

This instruction will cause the value of 4 to be added to the contents of $8 with the resulting value stored in $10. Note the instruction mnemonic. The “i” in addi stands for immediate signifying that this value is immediately available rather than having to be read from a register or memory. In this case the value can be encoded with the instruction using the I-format and thus accounting for the 16-bit immediate field in the instruction format. If the opcode for the addi instruction is 001000, using the I-format, the encoded value of the instruction addi $10, $8, 4 is 0x21480004. We can similarly conceive of immediate versions of all of the instructions shown in Table 1. For example, subi, sai, and sli.

Writing Programs

With the instructions defined so far we can write programs of the following form.

```
lw $10, $28
addi $28, $28, 4
lw $8, $28
add $9, $10, $8
addi $28, $28, 4
sw $9, $28
```

The preceding program loads two words from successive locations in memory, and computes the sum in $9 and stores the result back in memory. It is evident that writing programs comprised of such instructions is certainly preferable to writing microcode and is equivalent to describing computations using the RTL expressions presented at the beginning of this chapter.

Consider what we have learned so far. From the chapter on Memory Systems we can write sequences of data directives to store data in memory. We can also now write sequences of instructions that can process this data. Collectively these two program components permit us to write programs at the register transfer or *assembly language level*. At this level we deal with hardware entities such as registers and memory as opposed to variables and abstract data types that we find in high level languages such as Java or C. The programmer’s view of the machine is as a group of registers, a linear memory address space and a set of operations permitted using data stored in registers. If you are given a list of instructions and the operations they perform, you can be off and running writing interesting assembly language programs for all manner of computations. At this point the only capability missing is the ability to input data values from files or the keyboard and output data values to the files or the screen. Such operations are usually a function of the specific language and associated programming tools and can be found in documentation available with the specific programming tools being used. We will be using the SPIM

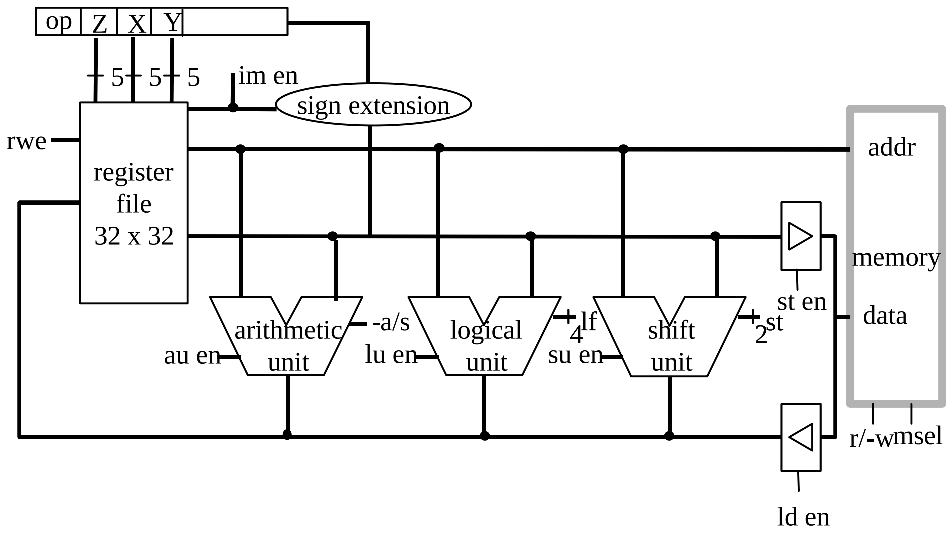

FIGURE 3. Single cycle datapath with an instruction register

simulator to execute programs written in SPIM, a subset of the MIPS R3000/4000 assembly language.

The set of instructions and the hardware entities that the instructions manipulate, such as registers, is collectively referred to as the *Instruction Set Architecture (ISA)*. We now continue with the development of the datapath to include the decoding of the assembly instructions and the logic required for the execution of programs comprised of assembly language instructions and data directives.

The Instruction Register and the Datapath

Since we would rather not write microinstructions, the assembly instructions form the interface between ourselves and the hardware. How are the encoded assembly instructions integrated into the datapath? How is the instruction encoding used to derive the control signals that implement the instruction? Solutions to these questions lead to the definition of the remaining components of the GT SPIM datapath.

Each instruction is encoded into a 32-bit pattern and stored in a special register referred to as the *instruction register (IR)*. The instruction register is integrated into the datapath as shown in Figure 3. The X, Y, and Z fields of the instruction are connected to the corresponding inputs of the register file. In other words, bits 25-21of the IR are connected to the Z address port of the register file, bits 20-16 of the IR are connected to the X address port of the register file, and bits 15-11 of the IR are connected to the Y address port of the register file. Therefore when an encoded R-format instruction is placed in the IR the X, Y, and Z fields of the instruction provide the addresses of the registers whose contents are placed on the X and Y buses as well as the address of the register which receives the contents of the result of the operation.

I-format instructions contain a 16-bit immediate value. This value serves as one of the operands and must be available on the Y bus. The ALU, logical unit and the shift units operate on 32-bit operands. The 16-bit immediate value must be extended to a 32-bit quantity. For positive numbers this is quite easy. We simply add leading zeros. For example, 0x1664 is equivalent to 0x00001664. However what if the number is a twos complement number where the most significant bit is 1 to signify a negative number? In this case the leading 16 bits that are added must be 1s (remember binary arithmetic!). Such an operation is referred to as *sign extension* since the sign bit of the 16-bit number is effectively "extended" to fill the remaining 16 bits. This relatively simple combinational logic block appears in Figure 3 as the sign extension unit. Consider an addi instruction. The 16-bit immediate field of the instruction now appears as a 32-bit operand on the Y bus. The imm en control signal enables the output of the sign extension unit while inhibiting the output of the register file that would otherwise drive the Y bus.

From the perspective of microcode the operation of the datapath has not changed. However, as a programmer it is now significantly easier to write programs although we are left with one problem. The instruction register can only hold one instruction at a time. Where do we store the large number of instructions that constitute a program? Why in memory of course! And how are these instructions transferred and placed in the IR? Instruction fetch logic is added to the datapath and we move one step closer to a complete datapath.

Next Instruction Logic and the Complete Single Cycle Datapath

Just as 32-bit data quantities are stored in memory 32-bit encoded instructions can also be stored in memory. We begin with the idea that we have a separate memory module for storing encoded instructions. Successive instructions are stored in contiguous words in memory. Given such an arrangement of instructions we are interested in designing the logic that will cause each of these instructions to be fetched and placed in the IR. A little thought will reveal the behavior of the instruction fetch logic. The first instruction must be loaded or fetched into the IR. After this instruction has been executed, the next

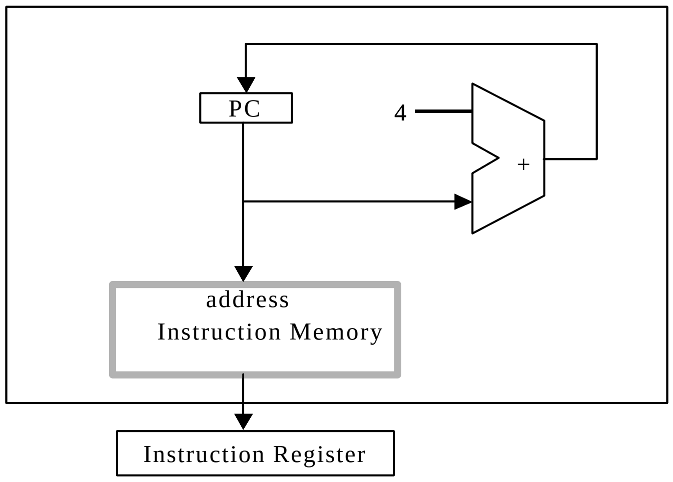

FIGURE 4. Instruction fetch logic

instruction should be fetched and executed. This process is repeated until all of the instructions have been executed. Implementation of this behavior can be realized as follows.

The datapath makes use of a special 32-bit register referred to as the *program counter* (PC). The program counter contains the address of the next instruction to be fetched and executed. Let us assume that the PC is initialized to the value 0x00000000 and we have a distinct memory module, instruction memory, that is used to store the instructions. While the datapath is operational the contents of the PC is sent to instruction memory and a read operation is asserted. The memory module returns the contents of location 0x00000000 which is the first encoded instruction. This instruction is stored in the instruction register. Now we wish to fetch the next instruction stored at location 0x00000004 (remember this is a byte addressed memory and each word take four bytes). The PC must be incremented by 4 and the process repeated to fetch the next instruction. This cycle of incrementing the PC by 4 and fetching instructions is repeated until the program terminates. The behavior is quite intuitive and the logic to implement this is shown in Figure 4. Each time and instruction is fetched the PC is incremented by 4 in preparation for the access of the next instruction. The implementation of the timing of the instruction fetch operation is described following the description of the controller.

This model of execution, *fetch-decode-execute*, is referred to as the von Neuman model in recognition of mathematician John von Neuman who is credited with his colleagues for the formulation of the model. Apart from special purpose machines the architectures of the vast majority of modern

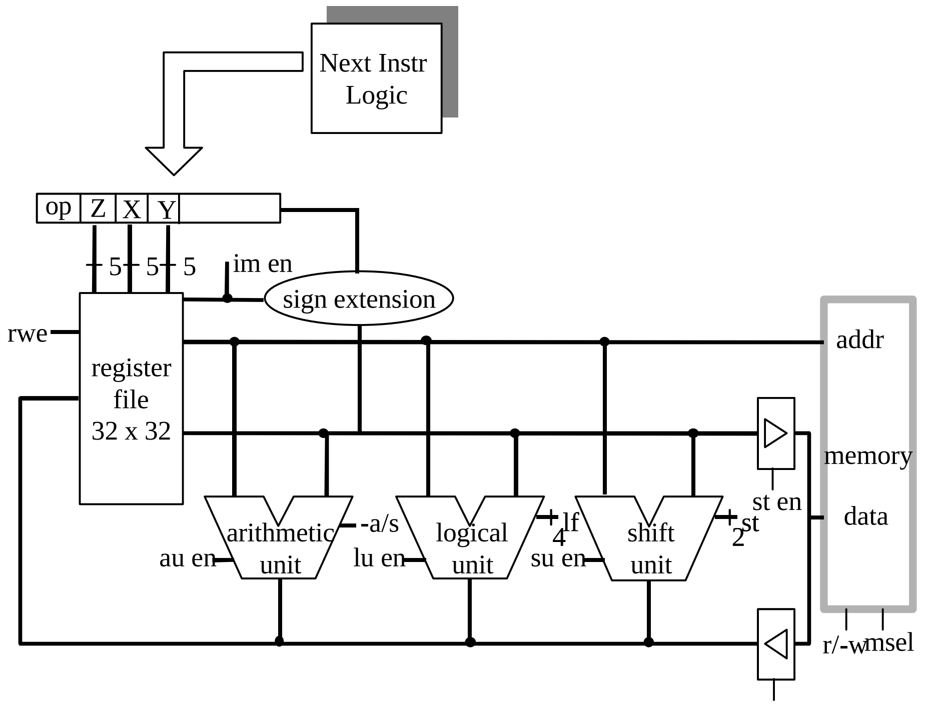

FIGURE 5. The single cycle datapath with instruction fetch logic

computers are derived from this basic model. The datapath integrated with the instruction fetch logic is shown in Figure 5.

The Controller

The remaining issue is the generation of the control signal values from the encoded instruction. The instruction opcode encodes the operation to be performed on the operands specified by the instruction. We require logic that decodes this opcode producing the corresponding control signals. This digital logic component that performs this decoding operation is referred to as a *controller* and will appear integrated into the datapath as shown in Figure 6.

As a prelude to this design we must provide a complete and unambiguous specification of the functionality of the control logic. Such a specification can be provided in a truth table. To simplify matters we

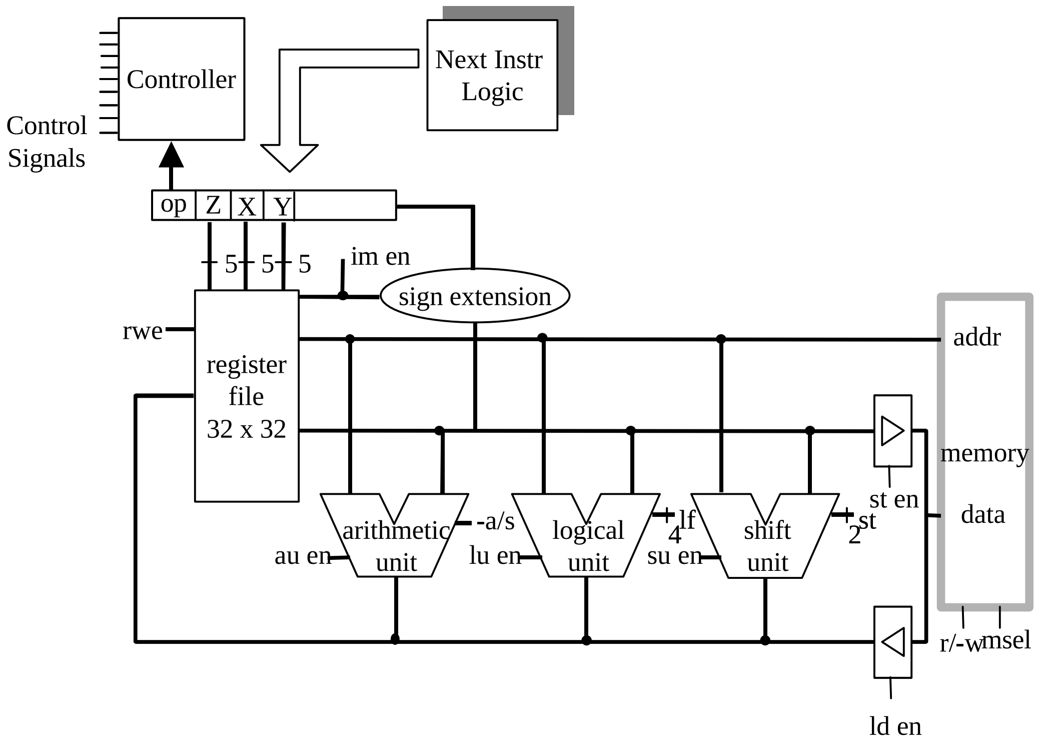

FIGURE 6. The complete single cycle datapath

will consider the design of a controller that supports only the instructions shown in Table 2. Associated with the instructions are shown the values of the opcodes that used for each instruction. Note the addition of the nop instruction. As the name suggests this instruction “does nothing”. It is often quite useful to have such an instruction and one application will be shown in an implementation of the controller.

Instruction	Opcode Value
add $10, $8, $9	100000
sub $10, $8, $9	100010
and $10, $8, $9	100100
or $10, $8, $9	100101
lw $10, $8	100011
sw $10, $8	101011
addi $10, $8, 4	001000
nop	000000

TABLE 2. **Opcode values for selected instructions**

With these opcode values and knowing the values of the control signals required to implement each instruction, the following truth table specification of the operation of the controller can be constructed as shown in Table 3. The first column represents the truth table inputs, namely the 6-bit opcode. The remaining columns provide the corresponding values of each control signal required to implement the instruction. With 6-bit inputs there are actually 2^6=64 rows for this truth table. However, we have only shown those rows that are relevant, that is, for the instructions from Table 2. These are the only instructions that we will implement for the example single cycle datapath. For the controller that we are designing the operation of the datapath is undefined for other opcode values.

Opcode (Instr)	rwe	imm en	au en	$\overline{\text{a}}$/s	lu en	lf	su en	st	st en	ld en	r/$\overline{\text{w}}$	msel
100000 (add)	1	0	1	0	0	0000	0	00	0	0	0	0
100010 (sub)	1	0	1	1	0	0000	0	00	0	0	0	0
100100 (and)	1	0	0	0	1	1000	0	00	0	0	0	0
100101 (or)	1	0	0	0	1	1110	0	00	0	0	0	0
100011 (lw)	1	0	0	0	0	0000	0	00	0	1	1	1
101011 (sw)	0	0	0	0	0	0000	0	00	1	0	0	1
001000 (addi)	1	1	1	0	0	0000	0	00	0	0	0	0
000000 (nop)	0	0	0	0	0	0000	0	00	0	0	0	0

TABLE 3. **Controller truth table**

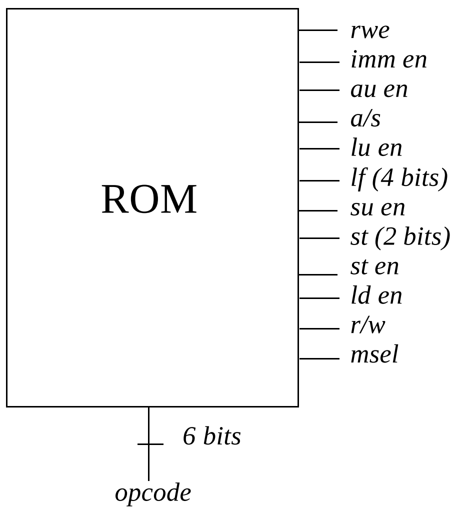

FIGURE 7. **Controller ROM**

There are several alternative implementations of this truth table. Two alternatives are discussed in the following.

Implementation 1: Read Only Memory (ROM)

Imagine a memory module designed such that a 16-bit value is stored at each address. Furthermore, consider a special type of memory from which values can only be read but not written. Such a memories are referred to as Read Only Memories (ROM). In this case values must be programmed into the memory module and once programmed can never be changed. We will use a ROM to store the control signal values shown in Table 3 in the following manner.

Consider the values of the 16 bits stored at memory address 0x100000. This address corresponds to the opcode signifying the addition operation. The contents of this memory location correspond to the 16 values in row 1 of Table 3. Similarly the values of the 16 bits stored at memory address 100100 correspond to the values in row 3 of Table 3. The operation of the ROM is functionally identical to that of a RAM. The opcode value is used as the address and the contents at that address correspond to the values of the 16 control signals required to implement the corresponding instruction. With a 6-bit opcode the ROM will have 64, 16-bit words. However not all of the memory locations will correspond to valid (according to Table 3) instructions. Those locations that do not correspond to valid instructions can be filled with values corresponding to the nop instruction.

```
.data
 .word 0x22
 .word 0x44

 .text
 lw $10, $28
 addi $28, $28, 4
 lw $9, $28
 add 11, $10, $9
 addi $28, $28, 4
 sw $11, $28
```

FIGURE 8. An example assembly language program

The opcode field of the instruction register can drive the address lines of the ROM. Each of the outputs of the ROM will be connected to the corresponding control signal in the datapath as shown Figure 7. As long as the ROM is correctly initialized to the correct values at each memory address this ROM will essentially serve as the mechanism for the translation of the opcode into control signal values. Each time a new instruction is loaded into the instruction register a new set of signals will be delivered to the datapath.

Implementation 2: Hardware Implementation

From the design of combinational logic systems the implementation of the truth table of Table 3 can be pursued via logic minimization with K-Maps and a hardware implementation using gates or programmable logic arrays (*material to be added*).

Putting It All Together

Now we can take a step back and with an example program take a look at the big picture. Let us start with the example program shown in Figure 8. This program is comprised of two main components. The first part is a set of data directives that describe the placement of data in memory. The data directives are preceded with a.data command to indicate to the assembler (reference Figure 1) that the following text is to be interpreted as data directives. The second component of the program is the assembly language instructions. These instructions are preceded by the.text command. The program simply reads two numbers from memory stored at consecutive locations and computes their sum. The sum is stored in mem-

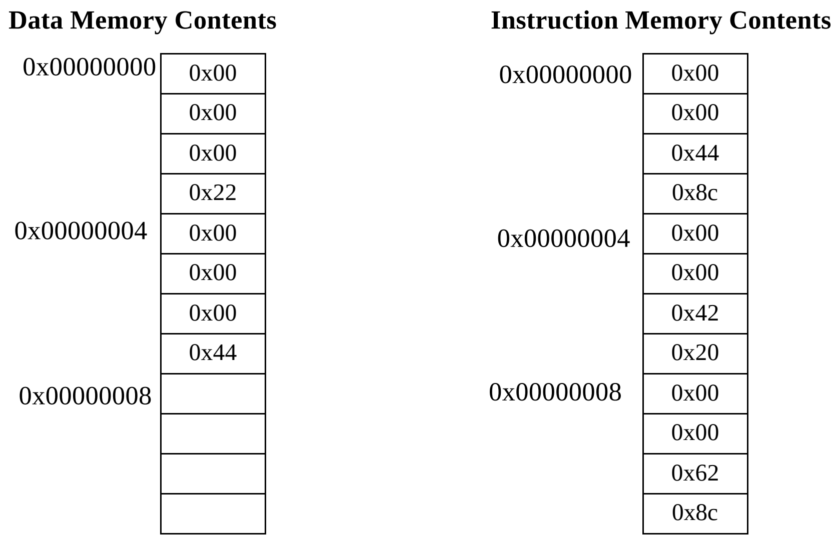

FIGURE 9. **Storage pattern resulting from the assembly of the program in Table 8**

ory at the third location. The first location in data memory and instruction memory is address is assumed to be 0x00000000 in this example. Let us start with the creation of this program and trace its behavior through execution on the datapath of Figure 6.

We might write such a program using a text editor and saving it to a file. Alternatively we may have written a simple program in C and the program shown in Figure 8 could have been generated by a compiler from this C program. In either case the textual representation of the program must be translated into the binary form suitable for storage and execution. This is the function of the *assembler*. An assembler will process the program in Figure 8 and encode each instruction into a 32-bit pattern according to the format definitions provided in Figure 2. Each data directive will be processed to cause the appropriate values to be loaded into data memory. For example, in the sample program the assembler will cause the value of 0x22 to be stored in data memory at address 0x00000000 and the value 0x44 to be stored in data memory at location 0x00000004. Each of the instructions will be stored in a single word at successive locations starting at address 0x00000000 in instruction memory. In practice another program called the loader will take the output of the assembler and perform the task of loading instruction and data memory. The memory contents resulting from this assembly procedure is shown in Figure 9. The assembler's job is now done we proceed to the execution of the program.

Let us assume that the PC is initialized to the value 0x00000000. This is the address that is sent to memory. The the 32-bit word from that address is fetched into the IR. This word is the encoding of the lw $10, $28 instruction. The opcode field of the instruction drives the address inputs to the controller ROM. The set of control signal values that are read from the ROM drive the datapath causing the contents of $28 (whose value is assumed to be initially 0x00000000) to be sent to the address input of data memory. The contents of this location are fetched and placed in $10. Now the program counter is incremented by 4 (refer to Figure 4) and the encoded value of the next instruction is fetched and placed in the IR. This instruction is an immediate instruction. The lower sixteen bits of this instruction contains the value 0x0004. This value forms the input to the sign extension unit which makes available the value 0x00000004 at the output of the sign extension unit. The opcode field will cause the corresponding control word to be fetched which will have the value of imm en equal to 1. This will cause the immediate value to be placed on the Y bus and form one of the two inputs to the adder/subtractor unit whose output is written to $28. The value of $28 is now 0x00000004. The following instruction will use the contents of $28 as an address into data memory which contains the value 0x00000044. The fetch-decode-execute process is repeated until the program execution is completed.

Datapath Timing

Consider how the single cycle datapath operates. An instruction is fetched from instruction memory and placed in the IR. The fields of this instruction now drive several datapath components. The X, Y, and Z fields drive the address inputs to the register file. The opcode drives the input to the controller and the lower 16 bits of the address drives the sign extension unit. How do each of the datapath components respond to their respective inputs?

The controller uses the opcode input to produce the values of all of the corresponding control signals. These values are available after a period of time corresponding to the delay to access the contents of the control ROM. Now each of the control signals can set the behavior of the corresponding datapath components. The X and Y address inputs of the register file cause the contents of these two registers to become available on the X and Y buses respectively where they can be processed by one of the individual datapath units with the resulting value placed on the Z bus if necessary (if it is not a memory store operation). The values of the register file control signals will determine if the Z bus value is to be written into the destination register. When all of this activity has ceased the next instruction can be fetched and the process is repeated.

Clearly the next instruction cannot be fetched until the execution of the previous instruction has been completed although we do wish to fetch it as soon as possible to ensure the highest possible rate of execution of instructions. The solution to the “when” an instruction can be fetched can be described with the respect to the next instruction logic in Figure 4. Note that whenever the PC is changed a new instruction is fetched. The PC is immediately incremented by 4 and this new value is available at the inputs to the PC. Since the IR is loaded as a consequence of loading the PC, a new instruction cycle starts with the loading of the PC with the next value. The loading of the PC can be controlled with the

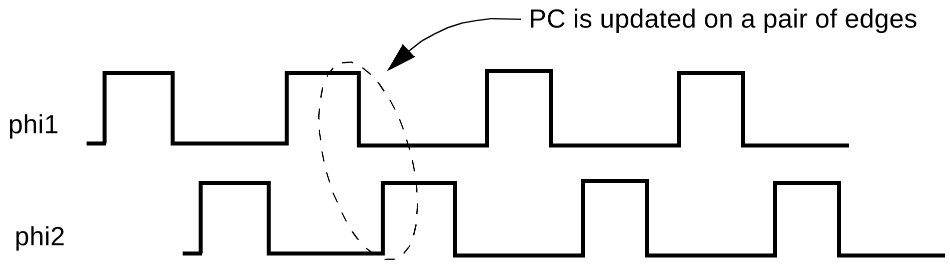

FIGURE 10. Two phase non-overlapping clocks for controlling the datapath

clock signals as portrayed in Figure 10. It is necessary to ensure that the clock periods are long enough to ensure that any instruction will complete execution prior to the loading of the PC with the address of the next instruction.

It is instructive to trace the execution of an instruction during a cycle, that is, between successive instruction fetch operations. Familiarity with the internal design of all of the datapath components makes this possible and leads to a detailed understanding of the datapath operation.

Instruction Set Architectures (ISA)

A programmers view of the datapath is comprised of the set of registers R0-R31 and the set of instructions that can operate on data in the registers as well as transfer data between registers and memory. The register set and instruction set do form a specification of the hardware that can serve as target for compilers and programmers. It can also serve as a starting point for computer architects and logic designers who would then proceed to design the hardware datapath that would execute programs written using this instructions set. This view of the datapath referred to as the Instruction Set Architecture or ISA.