

# Part III

## HARDWARE LEVELS

*By wisdom a house is built, and through understanding it is established; through knowledge its rooms are filled with rare and beautiful treasures.*

**Proverbs 24: 3-4**

---

The theme of this book is that a modern computer can be viewed as a series of abstract levels, each one implementing the one above it. The previous part of the book gave a perspective of the computer from the high-level language, assembly language, and machine language programmers' points of view. These views relate to the levels that typically deal with software. We now turn our attention to the levels that usually deal with hardware. The objective of the hardware is to interpret machine language programs. However, this interpretation itself is often done in several stages by different hardware levels. Although such a multi-layer interpretation may incur some performance cost, it does have important engineering advantages. The main advantage of adding an extra interpretation level is the ability to isolate implementation details of the underlying interpreter from those of the programs being interpreted. Once a language interface has been agreed upon, the respective development of interpretive and interpreted mechanisms can proceed more or less independently.

This part of the book focuses on the microarchitecture and the digital logic level, the levels that are immediately below the machine language level. The purpose of the microarchitecture level is to implement the ISA by interpreting ML programs, as illustrated in Figure 1.7 on page 34. The details of the microarchitecture depend on the ISA being implemented, the hardware technology available, and the cost and performance goals for the computer system. In Chapter 6, we discuss a simple microarchitecture for the entire computer. The objective is to present a microarchitecture that correctly implements the ISA and correctly interprets ML programs. The next 3 chapters focus on more advanced microarchitectures for each of the three major hardware sub-systems—the CPU, the memory system, and the IO system. In these discussions, we give special importance to achieving high performance and low power. One of the important microarchitectural techniques discussed for improving CPU performance is pipelining. For memory systems, we discuss cache memories. Finally, chapter 10 deals with the digital logic level.

---

## Chapter 6

# Microarchitecture Level — A System View

*Apply your heart to instruction and your ears to words of knowledge.*

**Proverbs 23: 12**

In this chapter, we study the organization and operation of the different components that constitute the microarchitecture of computers, the level that is responsible for implementing the instruction set architecture level—i.e., carrying out the execution of machine language programs. The microarchitecture is essentially an *emulator* for the machine language of the computer.

A particular ISA may be implemented in different ways, with different microarchitectures. The same executable program can be executed on these microarchitectures without any change, regardless of their differences, as long as they implement the same ISA. For example, the Intel IA-32 ISA has been implemented by Intel in different systems, such as Pentium and Pentium Pro. Apart from Intel, a number of competitors, such as AMD and Cyrix, have also developed many processors to implement the IA-32 ISA. A particular microarchitecture might focus on high performance, whereas another might focus on reducing the cost or the power consumption. The ability to develop different microarchitectures for the same ISA allows processor vendors to take advantage of new IC process technology, while providing upward compatibility to the users for their past investments in software. Although our discussion seems to indicate that the microarchitecture level is always implemented in hardware, strictly speaking, the microarchitecture level only specifies an abstract model. This abstract machine can be implemented in software if needed. An example is a cycle-accurate software *simulator*, such as the SimpleScalar simulator [ref].

Like the design of the higher levels, the design of the microarchitecture level is also replete with trade-offs, involving characteristics such as speed, cost, power consumption, die size, and reliability. For general-purpose computers, one trade-off drives the most important choices the microarchitect must make: speed versus cost. For laptops and embedded systems, the important considerations are size and power consumption. For space applications and other critical applications, reliability is of primary concern.

## 6.1 Overview of System Microarchitecture

As mentioned in the chapter's introduction, our next objective is to implement the machine specification given at the ISA level in hardware. Because of the complexity of this machine, it is impractical to directly design a gate-level circuitry that implements it. Therefore, in practice, we take a more structured approach by introducing one or more additional abstraction levels<sup>1</sup>. These levels are usually implemented in hardware. The abstraction level directly below the machine language level is the *microarchitecture level*, and is responsible for executing machine language programs.

Digital systems of reasonable complexity are often built using the *finite state machine* approach. After identifying a suitable *state* variable, the designer develops the state transition diagram, and implements the state transition diagram using appropriate set of flip-flops and logic gates. Theoretically, it is possible to build the computer hardware as a single finite state machine; after all, the computer hardware is a digital circuit. However, this finite state machine would have far too many states (a single bit change in a single memory location changes the system state), making it extremely difficult to comprehend the complex functionality, let alone design one in an efficient manner. Therefore, in practice, we take a different approach for designing the computer microarchitecture. First of all, we split the microarchitecture into two major parts—a **data path** and a **control unit**—as shown in Figure 6.1. The data path contains paths and circuitry required for carrying out data manipulation, and serve as a platform for executing ML programs. It is made up of storage components such as registers and memories, functional units such as ALUs, multipliers, and shifters, as well as interconnects to connect these components. The actions to be performed in a data path are specified using a language called *register transfer language (RTL)*. In order to execute ML programs on the data path, ML programs are interpreted in terms of RTL instructions. This interpretation is performed by the control unit, which passes the RTL instructions to the data path. Because a significant portion of the functionality has been shifted to the data path, the control unit can be conveniently built as a finite state machine with a manageable number of states. A change in a memory value does not cause a state transition in the control unit, for instance.

Both the data path and the control unit themselves are clocked sequential machines.

---

<sup>1</sup>Even for designing simple digital circuits, we do take a somewhat structured approach. When designing the circuit as a finite state machine, we first construct the state transition diagram, and then implement the state transition diagram.

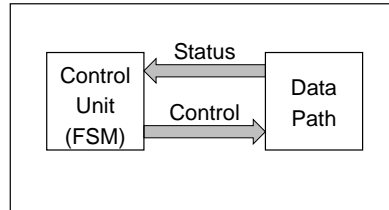


Figure 6.1: Designing the Computer Microarchitecture as a Combination of a Data Path and Control Unit

The sequence of operations performed in the data path is determined by *control inputs* generated by the control unit. The control unit thus functions as the interpreter of the machine language.

The computer data path can be easily divided into major building blocks and subsystems based on how data flow is specified in the ISA. Thus, we break the functionality specified in the machine language into different parts, and use separate building blocks to implement each of the parts in the data path. Thus at the system level, we see the computer hardware in terms of major building blocks and their interconnections.

The data path of a computer is built from building blocks such as registers, memory elements, arithmetic/logic units, other functional units, and interconnections. In order to design a data path for implementing an ISA we need to consider what the ISA requires for data flow through the system. There are two aspects to consider here:

- The storage locations (register name space, memory address space, and IO ports) defined in the ISA
- The operations or functions defined in the ISA.

## 6.2 Implementing the Storage Locations Defined in the ISA

We shall first take a look at implementing the storage locations defined in the ISA. These locations include the register name space, the memory address space, and the IO ports. All of these locations need to be implemented by appropriate storage elements in the data path.

The general-purpose registers specified in the ISA are typically implemented by a **register file (RF)**, which resembles a very small memory structure. Like a memory structure, the register file has an address input for specifying the register to be read from or written into. Thus, any specific register can be read or written by specifying the corresponding register number or address. An alternate approach, used especially when there are only a few general-purpose registers, is to implement each register separately. Apart from the

general-purpose registers, an ISA might include special registers such as PC, SP, and `flags`; floating-point registers such as F0 - F31; and privileged registers such as `status` register. These registers are also implemented using hardware registers in the data path.

The memory address space specified in the ISA can be implemented by a memory structure that accepts the address of a location, and performs a read/write to the specified location. A small portion of the memory address is typically implemented with non-volatile memory (to store the boot program), while the remaining portion is implemented with volatile memory for higher density.

The last category of storage locations specified in an ISA are the IO ports. Unlike the general-purpose registers and the memory locations, the IO ports are generally implemented in a distributed manner, using special register elements inside the different IO device interfaces. All of these storage elements are pictorially shown in Figure 6.2.

It is not sufficient to implement the storage locations using hardware storage elements. For meaningful operations to be carried out, it is important that the storage elements be connected properly.

By contrast, Figure 6.3 shows how the three categories of storage elements have been traditionally organized at the system level. We will see in Section 6.4 why we go for this kind of an organization.

## 6.3 Implementing the Functions Specified in the ISA

Once the storage locations defined in the ISA are implemented in the data path, the next step is to include additional elements to perform the operations and functionalities specified in the ISA. In order to do this, let us review the functions the data path must perform so as to execute machine language instructions. Some of the important functionalities are listed below.

- Data transfer between memory locations and registers.
- Data transfer between IO ports and registers
- Data transfer between registers
- Arithmetic/logical operations on data present in registers

### 6.3.1 Microarchitectural Registers

In order to carry out the above functionalities in a proper manner, the data path will need to have some additional internal memory to temporarily store information. For instance, it must store a copy of the ML instruction that is currently being executed in it; most data paths use a register called *instruction register* (IR) for this purpose. The data path may

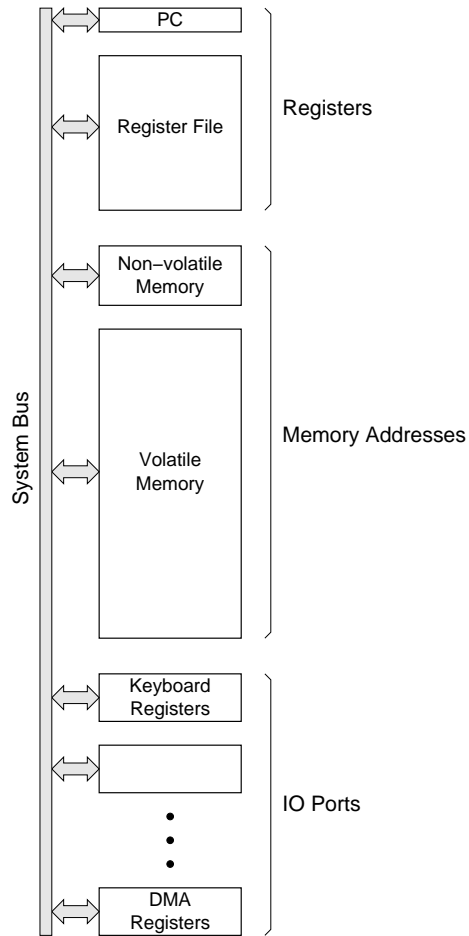


Figure 6.2: Implementing the Storage Locations Defined in the ISA

also need to keep track of special properties of arithmetic and logical operations such as condition codes; most data paths use a register called `flags`<sup>2</sup>. The data path may also need additional registers for temporarily storing data relevant to the currently executed instruction. Notice that all of these additional registers provided in the CPU data path are strictly microarchitectural, and therefore invisible to the machine language programmer.

<sup>2</sup>In some computers, the `flags` register is visible at the ISA level. In MIPS, it is not visible at the ISA level, whereas in IA-32 it is visible at the ISA level as a set of condition codes.

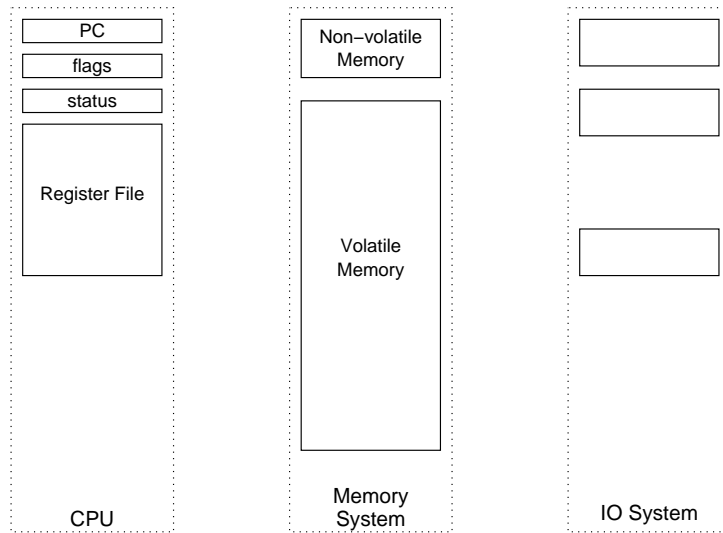
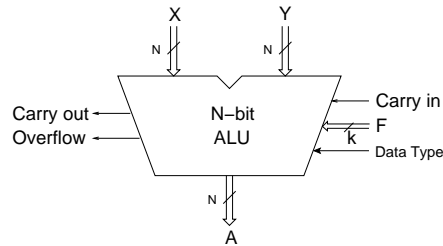


Figure 6.3: Implementing the Storage Locations Defined in the ISA — A System-Level View

### 6.3.2 ALUs and Other Functional Units

We have already considered the storage elements to be placed in the data path to perform its storage functions. Next, let us consider equipping the data path to perform arithmetic and logical operations, such as addition, subtraction, AND, OR, and shift. For carrying out these operations, we need to provide hardware circuitry that can perform different operations on bit patterns, and produce the correct bit patterns in the specific data encodings used. Many data paths consolidate the hardware circuitry for performing arithmetic and logical operations on integer data into a single multi-function block called *arithmetic and logic unit* (ALU), as shown in Figure 10.5. Hardware circuitry for performing complex operations such as integer multiplication, integer division, and all floating-point operations are usually not integrated into the ALU, and are built as separate *functional units*. This is because these operations typically require much longer times than that required for integer addition, integer subtraction, and all logical operations. Integrating all of these operations into a single ALU makes every operation as slow as the slowest one.

Figure 6.5 shows all of the storage elements (including the microarchitectural registers) along with the ALU.

Figure 6.4: An  $N$ -bit ALU Capable of Performing  $2^k$  Functions

### 6.3.3 Interconnections

Finally, the different blocks in a data path need to be interconnected so as to carry out the required data transfers. To achieve a reasonable speed of operation, most of the connections in the data path transfer a full word in parallel over multiple wires. Interconnects are either *bus-based* or *direct path-based*. A bus-based interconnect connects together a large number of blocks. At any particular instant, data can be transferred from a single block to one or more blocks. Thus it permits broadcasting of data to multiple destination blocks. In addition to the wires that carry the data, additional wires are included in a bus for *addressing* and *control* purposes. A direct path-based interconnect, on the other hand, relies on *point-to-point* connections between individual blocks. Such connections allow for *high speed*, but have *low versatility*. A typical data path has too many blocks for every block to be connected to every other block by point-to-point connections.

## 6.4 System Bus: Connecting the CPU, Memory, and IO Subsystems

We saw that at the system level, the computer microarchitecture contains 3 subsystems—the CPU, the memory subsystem, and the IO subsystem. For these subsystems to perform their respective system-level functions, they must be interconnected. For instance, the IO subsystem must be connected to the CPU and the memory subsystem in order for it to perform the IO functions. The subsystems can be interconnected in a variety of ways. The type of interconnects and the connectivity they provide determine, to a large extent, the time it takes to execute a machine language instruction. What would be a good interconnect to use here? The simplest type of connection that we can think of is to use a single bus to connect all of them, as illustrated in Figure 7.2. A bus is a shared communication link, connecting multiple devices using a single set of wires. Multiple devices can communicate over a single bus, at different times.

The two major advantages of the bus interconnect are *versatility* and *low cost*. Adding

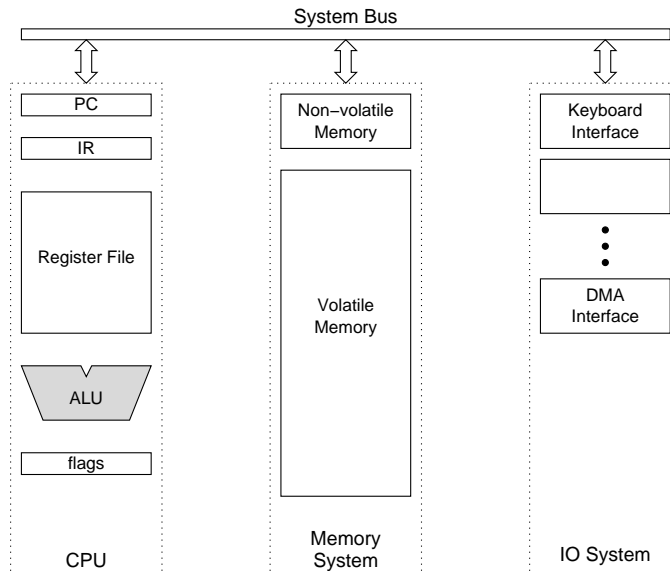


Figure 6.5: Implementing the Microarchitectural Registers and the ALU in the Computer Data Path

new devices is very straightforward. The major disadvantage of a bus is that it creates a communication bottleneck, possibly limiting the maximum IO throughput. When IO transfers must pass through a single bus, the bandwidth of that bus limits the maximum IO throughput.

The wires of the system bus can be grouped into three: address, data, and control. The address lines are used to send a memory address or an IO register address. The data lines are used to transfer the data value to be written or the value that is read from a location. The control lines are used to specify the nature of the transfer, and also to coordinate the actions of the sending and receiving units.

### 6.4.1 Multi-Bus System

In the discussion so far, the CPU, the memory subsystem, and the IO subsystem are interconnected with a single bus, called the *system bus*. Indeed such a connection is used in very small computers, where the CPU, the memory, and the IO interfaces are all physically placed on a single printed-circuit board (PCB), in which case the bus is fully contained within the board. The use of a single bus has several limitations:

- The specifications of a system bus (such as speed and word size) are defined primarily by the CPU requirements, and therefore tend to differ widely across different computer

families, and even between family members.

- The use of a single system bus permits only a single data transfer between the units at any given instant.
- In general-purpose computers, it is not possible to place all components on a single board.

Such systems comprise several circuit boards, typically housed in a card cage with connectors at the back (with the boards plugged into these connectors). The pins on different connectors are wired together to form a bus, called the *backplane bus* or *system bus*. Such a computer system may use the following types of buses: (i) *processor bus*, (ii) *processor-memory bus*, and (iii) *backplane bus*. Processor-memory buses are short, generally high speed, and matched to the memory system so as to maximize memory-processor bandwidth. Backplane buses are designed to allow the CPU, memory, and IO interfaces to coexist on a single bus; they balance the demands of processor-memory communication with the demands of IO interface-memory communication. Aside from the memory subsystem, the CPU talks to most of the remaining devices through the backplane bus. The backplane bus can be thought of as a major expressway that handles a large volume of high-speed traffic. In many recent machines, however, the distinction between backplane buses and processor-memory buses may be very minor.

#### 6.4.2 A Typical Desktop IO System

Figure 6.6 shows the system organization of a typical mid-range to high-end desktop machine in 2001. It uses a PCI bus as the backplane bus, with slower devices sharing the lower-performance SCSI bus. The PCI backplane bus is used to connect all interfaces to the processor and memory system. Several of the slow IO devices (audio IO, serial ports, and the desktop bus) share a single port onto the PCI bus. Serial ports provide for connections such as low-speed Appletalk network. The desktop bus provides support for keyboards and mice. The second interface in the figure is used for graphics output. The third interface is used for connecting to the ethernet network. The fourth interface is the SCSI bridge, which is used extensively for connecting the interfaces of bulk storage devices such as disks, tapes, and CD-ROMs. In the figure, a disk drive interface, a tape drive interface, and a CD-ROM interface are connected to the SCSI bus.

### 6.5 Instruction-Level Simulator: Software Microarchitecture

The discussion in this chapter may inadvertently seem to imply that microarchitectures for executing machine language programs are by nature hardware-oriented. However, this is not the case. As pointed out in chapter 1, the distinction between hardware and software is

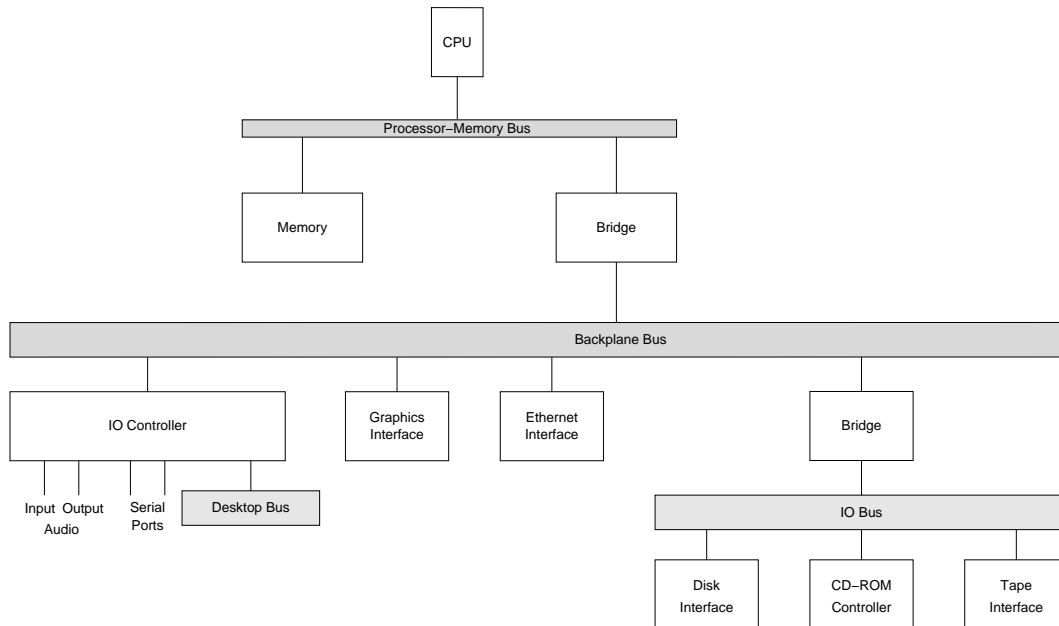


Figure 6.6: System Organization in a Typical Desktop Computer

quite blurd. This means that we can implement a computer microarchitecture in software; such a software-oriented microarchitecture is frequently called a *software simulator*.

A software simulator program can perform its emulation function only when it is executed on another microarchitecture (host machine), entailing an interpretation by the host machine's control unit. Thus, implementing a microarchitecture in software introduces an extra interpretation step for the ML program. Why in any case, would anyone want to introduce extra interpretation steps? Although it may not make much sense for ordinary users to take this route, software simulation is the de facto tool of the trade for computer architects and microarchitects, who are in the business of developing new ISAs and microarchitectures.

Software simulators can be implemented at different levels of abstraction. An instruction-level simulator (sometimes called a *functional simulator*) is one that correctly executes ML programs, without modeling any hardware devices. Such a simulator has no notion of hardware-specific features such as clock cycles and buses. A cycle-accurate simulator, on the other hand, models a specific hardware microarchitecture, and knows about clock cycles and buses. In fact, it simulates the hardware to clock cycle accuracy; hence its name. Because of this extra level of details, the cycle-accurate simulator is considerably more complex and slower than an instruction-level simulator.

## Chapter 7

# CPU Microarchitecture

*Go to the ant, you sluggard; consider its ways and be wise! It has no commander, no overseer or ruler, yet it stores its provisions in summer and gathers its food at harvest.*

**Proverbs 6: 6-8**

The central processing unit (CPU) plays a central role in a computer microarchitecture's function. It communicates with and controls the operation of other subsystems within the computer. Much of the discussion in this chapter deals with the microarchitecture of the CPU. The performance of the CPU has a major impact on the performance of the computer system as a whole. So we also study advanced microarchitectural techniques for improving CPU performance.

This chapter addresses some of the fundamental questions concerning CPU microarchitecture, such as:

- What are the building blocks in a CPU data path, and how are they connected together?
- What steps should the CPU data path perform to sequence through a machine language program, and to execute (i.e., accomplish the work specified in) each machine language instruction?
- What are some simple organizational techniques to reduce the number of clock cycles taken to execute each machine language instruction?

## 7.1 A Simple CPU Data Path for Executing MIPS ML Subset Programs

Although CPU data paths can be designed in a generic manner to support a variety of ISAs, such an approach is rarely taken. In practice, each CPU data path is designed with a specific ISA in mind<sup>1</sup>. The primary reason for this is performance. If we design a generic CPU data path to support the idiosyncrasies of a number of ISAs, then that data path is likely to be significantly slower than one that caters to a specific ISA. Therefore, we will first discuss a simple example data path in detail, and then move on to more complex data paths. This simple data path will be used in the discussions of the control unit as well. For the sake of continuity with the preceding chapters, this data path is designed for executing MIPS ML instructions. For simplicity and ease of understanding, we restrict ourselves to a subset of the MIPS ML instruction set. This subset, along with the encoding formats used by the subset, is given in Figure 7.1.

In order to design a data path for this MIPS instruction subset, first consider the ISA-defined storage locations that are implemented within the CPU. These include the general-purpose registers and the special registers. The MIPS ISA defines 32 general-purpose registers, R0 - R31, and we use a 32-entry register file to implement them. In each clock cycle, this register file can accept a single address input for specifying the register to be read from or written into. This data path uses the special register PC to store the memory address of the next instruction to be interpreted and executed.

Apart from the ISA-visible registers, we include the following microarchitectural registers:

- **IR**: Stores a copy of the current ML instruction (bit pattern) being executed
- **AIR**: Stores one of the inputs to the ALU.
- **AOR**: Stores the result produced by the ALU.
- **flags**: Stores important properties of the result produced by the ALU. Bits of this register can be accessed individually.

We shall use a single multi-function ALU (Arithmetic and Logic Unit) to perform the arithmetic and logical operations specified by the ML instruction subset. Because most of the ALU operations work with values stored in registers, the ALU is typically included in the CPU data path.

Figure 7.2 shows one possible way of interconnecting the main building blocks so as to perform the required functions. This figure suppresses information on how the control unit controls the functioning of the blocks; these details will be added later. For getting a better

---

<sup>1</sup>This is in contrast to the practice followed in designing an ISA, where the design is not tailored for any particular high-level language.

LUI	rt, immed	001111		rt		immed
LW	rt, offset(rs)	100011	rs	rt		offset
SW	rt, offset(rs)	101011	rs	rt		offset
ADDI	rt, rs, immed	001000	rs	rt		immed
ADD	rd, rs, rt	000000	rs	rt	rd	100000
SUB	rd, rs, rt	000000	rs	rt	rd	100010
ANDI	rt, rs, immed	001100	rs	rt		immed
AND	rd, rs, rt	000000	rs	rt	rd	100100
ORI	rt, rs, immed	001101	rs	rt		immed
OR	rd, rs, rt	000000	rs	rt	rd	100101
NOR	rd, rs, rt	000000	rs	rt	rd	100111
SLLV	rd, rs, rt	000000	rs	rt	rd	000100
BEQ	rs, rt, offset	000100	rs	rt		offset
BNE	rs, rt, offset	000101	rs	rt		offset
JALR	rd, rs	000000	rs		rd	001001
JR	rs	000000	rs			001000
SYSCALL		000000				001100
RFE		010000	1			100000

Figure 7.1: A Subset of the MIPS Instruction Set, along with the Encoding Used.

overall picture, the figure also shows the main memory system and the IO system, which are traditionally placed outside the CPU chip<sup>2</sup>. This simple data path uses a single CPU internal bus (32 bits wide) for connecting the registers and other blocks inside the CPU. In addition to the bus, we have some dedicated paths to perform transfers that are inefficient to be mapped to the bus. One such path connects IR to the **sign extend** unit. The use of a bus permits a full range of paths to be established between the blocks connected by the bus. It is quite possible that some of these paths may never have to be used. For instance, there is no need to copy the contents of PC to IR. The direct path based data path that we will see later in this chapter eliminates all such unnecessary paths by providing independent

<sup>2</sup>As we will see in Chapter 8, the memory system microarchitecture of a typical computer includes multiple levels of cache memories, of which the top one or two levels are often integrated into the CPU chip. With continued advancements in VLSI technology, a number of researchers are even working towards the integration of the CPU and the memory system into a single chip.

connections only between those blocks that need to communicate.

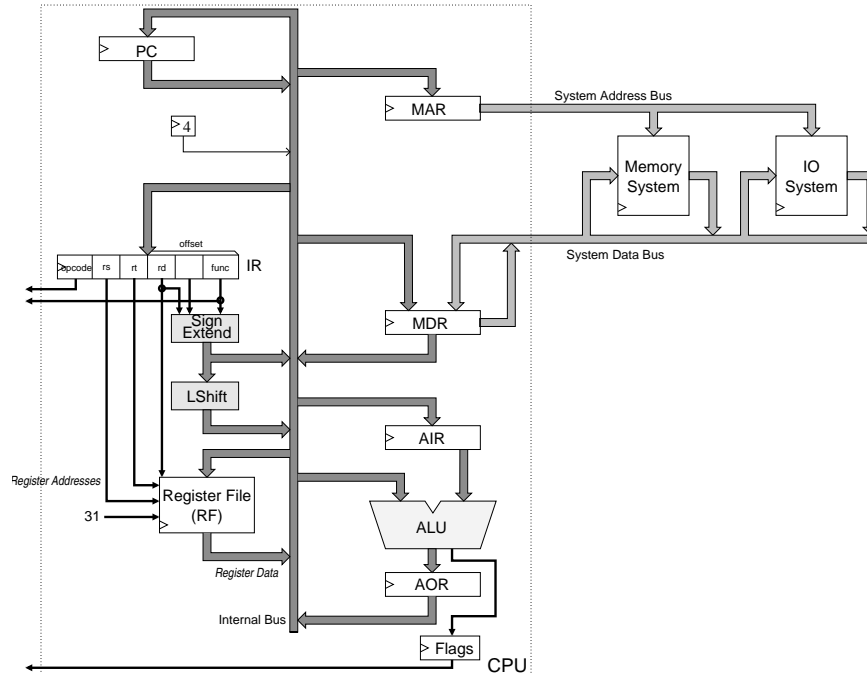


Figure 7.2: A Simple Data Path with a Single CPU Internal Bus

The data path uses the microarchitectural register **IR** to store the bit pattern of the instruction being interpreted and executed. Unlike other registers, the outputs from **IR** are organized as *fields*, in line with the different fields specified in the MIPS instruction formats. The different outputs of **IR** are illustrated in Figure 7.3. The **opcode** and **func** outputs are supplied to the CPU control unit (not shown in the figure). The **rs**, **rt**, and **rd** outputs are used to select the register that is to be read or written into. The register read addresses can be **rs** or **rt**, whereas the register write address can be **rt**, **rd**, or **31**. (Recall that when a subroutine call instruction is executed, the return address is written into **R31**.) The **offset** output includes the least significant 16 bits of **IR**, and is supplied to a **Sign Extend** unit to convert it into a 32-bit signed integer. This 32-bit output of the **Sign Extend** unit is connected to the CPU's **Internal Bus** directly as well as through a **LShift** unit that performs a left shift by two bit positions, so as to multiply it by 4. The former connection is provided for the purpose of executing the memory-referencing instructions and the register-immediate ALU type instructions. The latter is provided for executing the branch-type instructions, which involve multiplying the **offset** value by 4.

For completeness, the figure also shows the memory subsystem and the IO subsystem, which implement the memory address space and IO address space, respectively, that are

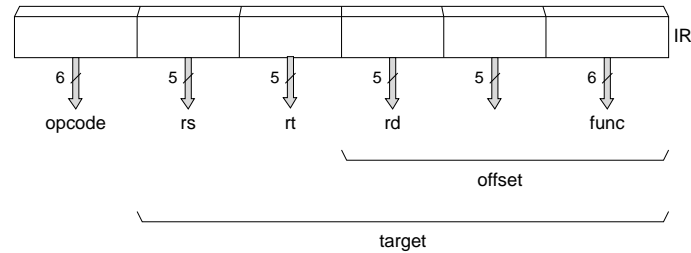


Figure 7.3: Different Outputs of Microarchitectural Register IR, which holds a copy of the Instruction Bit Pattern

defined in the ISA. Finally, for interfacing the processor bus to the memory system and the IO system, many CPU data paths provide two special microarchitectural registers. The standard convention is to call them *memory address register (MAR)* and *memory data register (MDR)*. The former is used to store the address of the memory location or IO port to be read from or written to, and the latter is used to store the data that is read or to be written. The address stored in MAR is made available to the address inputs of the memory system and the IO system through the **System Address Bus** during a memory/IO read/write operation. When a memory/IO read operation is performed, MDR is updated from the **System Data Bus**. Similarly, when a memory/IO write operation is performed, the contents of MDR are transmitted to the appropriate memory/IO unit through the **System Data Bus**.

## 7.2 Microassembly Language Instructions

Each machine language instruction can be executed at the microarchitectural level using a sequence of elementary instructions on an appropriate data path. We shall represent these microarchitectural-level elementary instructions using mnemonics (similar to what is done in assembly languages as studied in chapters 3 and 4). Accordingly, we shall call these elementary instructions *microassembly language ( $\mu$ AL) instructions*, to indicate their resemblance to the assembly language instructions and to indicate that they are microarchitecture-level instructions. A  $\mu$ AL instruction consists of one or more elementary operations called  *$\mu$ AL operations*, performed on data stored in registers or in memory. A  $\mu$ AL operation can be as simple as copying data from one physical register to another, or more complex, such as adding the contents of two physical registers and storing the result in a third physical register. An example  $\mu$ AL operation is

$$\text{PC} \rightarrow \text{MAR}$$

### 7.2.1 Microassembly Language ( $\mu$ AL): A Language to Specify Microassembly Instructions

Next we will discuss a special language called microassembly language for specifying the elementary microarchitecture-level instructions. In this language, data transfer operations and arithmetic/logical operations are specified using a language called *register transfer language (RTL)*. In RTL, data transfer is designated in symbolic form by means of the replacement operator ( $\rightarrow$ ). Thus, the notation we use to specify the  $\mu$ AL operation that copies the contents of PC to MAR is

$$\text{PC} \rightarrow \text{MAR}$$

By definition, the contents of the source register do not change as a result of the transfer. The  $\mu$ AL operation does not specify how data is copied from PC to MAR; it merely specifies what transfer needs to be made. The specifics of the data transfer are dealt with at the digital logic level. Notice that

$$\text{IR} \rightarrow \text{AOR}$$

is not a valid  $\mu$ AL operation in the data path given in Figure 7.2, because there is no direct connection between IR and AOR. Similarly,

$$\text{IR} \rightarrow \text{ALU}$$

is also not a valid  $\mu$ AL operation, because ALU is combinational logic, and not a storage device.

Normally, all bits of a register are involved in a transfer. However, if a subset of the bits is to be transferred, then the specific bits are identified by the use of pointed brackets. The  $\mu$ AL operation

$$\text{IR}\langle 15:0 \rangle \rightarrow \text{MDR}$$

specifies that bits 15 to 0 of IR are transferred to MDR. Similarly, memory locations or general-purpose registers (GPRs) are specified with square brackets. The  $\mu$ AL operation

$$\text{R}[\text{rs}] \rightarrow \text{MDR}$$

indicates that the contents of the GPR whose address is present in the *rs* field (of IR) are transferred to MDR. The *rs* field specifies a particular GPR. Similarly, the  $\mu$ AL operation

$$\text{MDR} \rightarrow \text{M}[\text{MAR}]$$

indicates that the contents of MDR are transferred to the memory location whose address is present in MAR.

If the interconnections of the data path are rich enough to allow multiple  $\mu$ AL operations in the same time period, these can be denoted by writing them in the same line, as follows:

$$\text{PC} + 4 \rightarrow \text{AOR}; \quad \text{M}[\text{MAR}] \rightarrow \text{MDR}$$

specifies that in the same time period, the value of PC is incremented by 4 and written to AOR, and the contents of memory location addressed by MAR are transferred to MDR.

Finally, for  $\mu$ AL operations that are conditional in nature, an “if-else” construct patterned after the C language’s “if-else” construct is provided.

```

if (C == 1)  SE(Offset) + AIR  → AOR
             else  4 + AIR     → AOR

```

indicates that if the carry flag is equal to 1, the `Offset` field (of `IR`) is sign extended, added to the contents of `AIR`, and the result is stored in `AOR`; otherwise 4 is added to `AIR`, and the result is stored in `AOR`.

### 7.2.2 $\mu$ AL Operation Types

The actions performed by  $\mu$ AL operations can be classified into three different types:

1. *Data transfer*  $\mu$ AL operations copy the contents of one register/memory location to another. An example data transfer  $\mu$ AL operation is `PC → MAR`.
2. *Arithmetic/Logic*  $\mu$ AL operations perform arithmetic or logical operations on data stored in registers. Data transfers can only occur along the interconnections provided in the data path, from one register/memory location to another. An example arithmetic  $\mu$ AL operation is `PC + AIR → AOR`.
3. *Control changing*  $\mu$ AL operations, which perform a control flow change within the  $\mu$ AL instruction sequence. An example control changing  $\mu$ AL operation is `goto state n`.

A given  $\mu$ AL instruction may specify actions of more than one type. At the microarchitecture level, several  $\mu$ AL instructions are grouped together to form sequences that interpret different machine language instructions.

## 7.3 Interpreting ML Programs by $\mu$ AL Routines

Having discussed the basics of a data path, the next step is to investigate how machine language programs can be interpreted by a set of  $\mu$ AL instructions defined for a particular data path. This section discusses how  $\mu$ AL instructions can be put together to carry out this interpretation. The interpretation process can be divided into two parts:

- Sequencing through the ML program and obtaining the next ML instruction to be interpreted
- Interpreting the next ML instruction

We will see how each of these parts can be accomplished by a sequence of  $\mu$ AL instructions called a  $\mu$ AL routine. To facilitate the interpretation of ML programs in the proper manner, a  $\mu$ AL routine is allowed to freely use and modify any of the ISA-invisible microarchitectural registers. The ISA-visible registers, however, can be modified only as per the semantics of the ML instruction being interpreted.

In order to do this, let us review the functions the data path must perform so as to execute machine language instructions. The interpretation of an ML instruction can be done in a sequence of 5 phases:

- 
1. *Fetch instruction*: Read the next instruction from the main memory into the CPU.
  2. *Decode instruction*: Decode the instruction bit pattern so as to determine the action specified by it.
  3. *Fetch source operand values*: Fetch source operands (if any) from registers, main memory, or IO registers.
  4. *Process data*: Perform arithmetic or logical operation on source operand values, if required.
  5. *Write result operand value*: Write the result of an arithmetic or logical operation to a register, memory location, or IO register, if required.
- 

These 5 phases enable a data path to carry out the execution of a single ML instruction. In order to execute an ML program, which is a sequence of ML instructions, the data path should also implement the *sequencing* among ML instructions. That is, after carrying out the above 5 phases to execute a single ML instruction, it should go back to phase 1 to begin executing the next instruction in the ML program.

### 7.3.1 Interpreting an Arithmetic/Logical ML Instruction

First let us consider an arithmetic/logical ML instruction. We shall put together a sequence of  $\mu$ AL instructions to interpret an arithmetic/logical instruction.

*Example:* Consider the MIPS ADD instruction whose symbolic representation is `addu rd, rs, rt`. Its encoding is given below.

000000	rs	rt	rd		ADD
--------	----	----	----	--	-----

Table 7.1 specifies a sequence of  $\mu$ AL instructions for fetching and executing this instruction in the data path given in Figure 7.2. Let us go through the working of this  $\mu$ AL

Step No.	$\mu$ AL Instruction	Comments
	for Data Path	
<i>Fetch and Decode phase</i>		
0	PC $\rightarrow$ MAR	<i>Decode instr</i>
1	M[MAR] $\rightarrow$ MDR	
2	MDR $\rightarrow$ IR	
3		
<i>Execute phase</i>		
4	R[rs] $\rightarrow$ AIR	
5	R[rt] + AIR $\rightarrow$ AOR	
6	AOR $\rightarrow$ R[rd]	
<i>PC increment phase</i>		
7	PC $\rightarrow$ AIR	<i>Goto step 0</i>
8	AIR + 4 $\rightarrow$ AOR	
9	AOR $\rightarrow$ PC	

Table 7.1: A  $\mu$ AL Routine for Interpreting the MIPS ML Instruction Represented Symbolically as `addu rd, rs, rt` for Execution in the Data Path of Figure 7.2

routine. The first phase involves fetching the instruction bit pattern from memory into the CPU microarchitectural register IR.

Recall that the address of the machine language instruction to be interpreted is already present in PC. The first step therefore involves copying the contents of PC to MAR so that this address can be supplied through the **System Address Bus** to the memory system. Thus, at the end of step 0, MAR has the address of the memory location that contains the instruction bit pattern. Step 1 performs a memory read operation, during which the contents of the memory location specified through the **System Address Bus** by MAR are read from the memory. The instruction bit pattern so read is placed on the **System Data Bus**, from where it is loaded into the CPU's MDR. Thus, at the end of step 1, the instruction bit pattern is present in MDR. In this microroutine, we consider the time taken to perform this step as one clock cycle, although the exact number of cycles taken depends on the specifics of the memory system used. In step 2, the instruction bit pattern is copied from MDR into IR.

Once the instruction bit pattern is copied into IR, the instruction decoding circuitry interprets the contents of IR. This decode process takes place in step 3, and enables the control circuitry to choose the appropriate microassembly-instructions for the remainder of the interpretation, steps 4-6, which constitute the execution phase. Steps 0-3 constitute the instruction fetch and decode phases. Naturally, this portion is the same for every machine language instruction in the MIPS ISA because of its fixed length instruction formats. Had the MIPS ISA used variable length instructions, this fetch process would have to be repeated as many times as the number of words in the instruction.

In step 4, the contents of the `rs` field of IR are used to read general-purpose register numbered `rs` into microarchitectural register AIR. In step 5, the contents of general-purpose register numbered `rt` are read and supplied to the ALU, which adds it to the contents of AIR, and stores the result in the microarchitectural register AOR. In step 6, the contents of AOR are transferred to general-purpose register numbered `rd`. By performing this sequence of  $\mu$ AL instructions in the correct order, the ML instruction `addu rd, rs, rt` is correctly interpreted.

After interpreting an ML instruction, the interpreter needs to go back to step 0 to interpret the next instruction in the ML program. However, prior to that, it has to increment PC to point to the next instruction; otherwise the same ML instruction gets interpreted repeatedly. In steps 7-9, PC is incremented by 4 to point to the next instruction in the executed machine language program. After step 9, the interpreter goes back to step 0.

This  $\mu$ AL routine takes 10 clock cycles to complete. We can, in fact, perform the PC increment phase in parallel to the instruction fetch phase, and reduce the total number of clock cycles required for the interpretation. Table 7.2 provides the modified  $\mu$ AL routine, which requires only 7 clock cycles. In this routine, in steps 0 and 1, the updated value of PC is calculated in parallel with the transfer of the instruction bit pattern from the main memory to MDR. It is important to note that multiple  $\mu$ AL operations can be done in parallel, only if they do not share the same bus or destination register. In general, the more the connectivity provided in a data path, the more the opportunities for performing multiple  $\mu$ AL operations in parallel.

Step No.	$\mu$ AL Instruction for Data Path		Comments
	<i>Fetch and Decode phase</i>		
0	PC	$\rightarrow$ MAR; PC $\rightarrow$ AIR	
1	M[ <code>MAR</code> ]	$\rightarrow$ MDR; AIR + 4 $\rightarrow$ AOR	
2	MDR	$\rightarrow$ IR	
3	AOR	$\rightarrow$ PC	
<i>Execute phase</i>			<i>Decode instr</i>
4	R[ <code>rs</code> ]	$\rightarrow$ AIR	<i>Goto step 0</i>
5	R[ <code>rt</code> ] + AIR	$\rightarrow$ AOR	
6	AOR	$\rightarrow$ R[ <code>rd</code> ]	

Table 7.2: An Optimized  $\mu$ AL routine for Interpreting the MIPS ML Instruction Represented Symbolically as `addu rd, rs, rt` for Execution in the Data Path of Figure 7.2

### 7.3.2 Interpreting a Memory-Referencing ML Instruction

Let us now put together a sequence of  $\mu$ AL instructions to fetch and execute a memory-referencing machine language instruction. Because all MIPS instructions are of the same length, the  $\mu$ AL routine for the fetch part of the instruction is the same as before; the differences are only in the execution part. Consider the MIPS load instruction whose symbolic representation is `lw rt, offset(rs)`. The semantics of this instruction are to copy to GPR `rt` the contents of memory location whose address is given by the sum of the contents of GPR `rs` and sign-extended `offset`. We need to come up with a sequence of  $\mu$ AL instructions that effectively fetch and execute this instruction in the data path given in Figure 7.2.

100011	rs	rt	offset
--------	----	----	--------

The interpretation of a memory-referencing instruction requires the computation of an address. For the MIPS instruction set, address calculation involves sign-extending the `offset` field of the instruction to form a 32-bit signed offset, and adding it to the contents of the register specified in the `rs` field of the instruction. In this data path, the address calculation is done using the same ALU, as no separate adder has been provided. With this introduction, let us look at the  $\mu$ AL routine given in Table 7.3 to interpret this `lw` instruction.

Step No.	$\mu$ AL Instruction for Data Path	Comments
<i>Fetch and Decode phase</i>		
0	PC $\rightarrow$ MAR; PC $\rightarrow$ AIR	
1	M[MAR] $\rightarrow$ MDR; AIR + 4 $\rightarrow$ AOR	
2	MDR $\rightarrow$ IR	
3	AOR $\rightarrow$ PC	<i>Decode instr</i>
<i>Execute phase</i>		
4	R[rs] $\rightarrow$ AIR	
5	SE(offset) + AIR $\rightarrow$ AOR	
6	AOR $\rightarrow$ MAR	
7	M[MAR] $\rightarrow$ MDR	
8	MDR $\rightarrow$ R[rt]	<i>Goto step 0</i>

Table 7.3: An Optimized  $\mu$ AL Routine for Interpreting the MIPS ML Instruction Represented Symbolically as `lw rt, offset(rs)` for Execution in the Data Path of Figure 7.2

### 7.3.3 Interpreting a Control-Changing ML Instruction

The instructions that we interpreted so far do not involve control flow changes that cause deviations from straightline sequencing. Next let us see how our data path can be used to interpret control-changing instructions, which involve modifying PC, usually based on a condition.

*Example:* Consider the MIPS ML conditional branch instruction whose symbolic representation is `beq rs, rt, offset`. The semantics of this instruction state that if the contents of GPRs `rs` and `rt` are equal, then the value `offset × 4 + 4` should be added to PC so as to cause a control flow change<sup>3</sup>; otherwise, PC is incremented by 4 as usual. The encoding of this instruction is given below:

000100	rs	rt	offset
--------	----	----	--------

Table 7.4 presents a  $\mu$ AL routine to fetch and execute this instruction in the data path given in Figure 7.2. Again, the routine for the fetch part is same as that of the previous instructions.

Step No.	$\mu$ AL Instruction for Data Path	Comments
<i>Fetch and Decode phase</i>		
0	PC $\rightarrow$ MAR; PC $\rightarrow$ AIR	
1	M[MAR] $\rightarrow$ MDR; AIR + 4 $\rightarrow$ AOR	
2	MDR $\rightarrow$ IR	
3	AOR $\rightarrow$ PC	<i>Decode instr</i>
<i>Execute phase</i>		
4	R[rs] $\rightarrow$ AIR	
5	R[rt] == AIR $\rightarrow$ Z	
6	LS(SE(offset)) $\rightarrow$ AIR	
7	PC + AIR $\rightarrow$ AOR	
8	if (Z) AOR $\rightarrow$ PC	<i>Goto step 0</i>

Table 7.4: An Optimized  $\mu$ AL Routine for Interpreting the MIPS ML Instruction Represented Symbolically as `beq rs, rt, offset` for Execution in the Data Path of Figure 7.2

In the execute routine, the interesting  $\mu$ AL instructions are those of steps 5, 6, and 8. In step 5, the contents of register `rt` are compared against those of AIR (which were copied from register `rs`), and flag Z is set if they are the same. In step 6, the output of the LS unit,

<sup>3</sup>The actual MIPS ISA uses a *delayed branch* scheme; i.e., the control flow change happens only after executing the next ML instruction in the program.

which is 4 times the sign-extended `offset`, is copied to `AIR` for adding to the updated `PC` in the next step. In the last step, `PC` is updated with the calculated target value present in `AOR`, if flag `Z` is 1. Thus, if flag `Z` is not set, then `PC` retains the updated value it obtained in step 3, which is the address of the fall-through instruction in the machine language program being interpreted.

## 7.4 CPU Control Unit: Interpreter for Machine Language Programs

We have seen how individual machine language instructions can be interpreted in a data path using a sequence of  $\mu$ AL instructions. Next we will look into the hardware unit that is responsible for generating the  $\mu$ AL instructions in the proper order. This is the *CPU control unit*. It is, in some sense, the CPU's brain, and is often the most complex part of the design of computers. The control unit keeps track of the state of the processor on a cycle-by-cycle basis, and generates in each cycle the required  $\mu$ AL instruction, in an encoded form called **microinstruction**<sup>4</sup>. An important note is in order here. A microarchitect typically uses  $\mu$ AL instructions and  $\mu$ AL routines when specifying how a data path should interpret each ML instruction. These  $\mu$ AL instructions and  $\mu$ AL routines are translated into binary encoded **microinstructions** and **microroutines** by hand or by a **microassembler**. It is more efficient for the control unit to directly generate the binary encoded microinstructions, rather than generate the English-like  $\mu$ AL instructions, and then translate them into microinstructions. The microassembler is similar to an assembler.

Figure 7.4 shows the general relationship between the data path and the CPU control unit. Every clock cycle, the CPU control unit receives the status of the data path, and specifies the microinstruction to be performed by the data path in that cycle. Thus, the CPU control unit controls the processing of data within the CPU data path, as well as the flow of data within, to, and from the CPU data path.

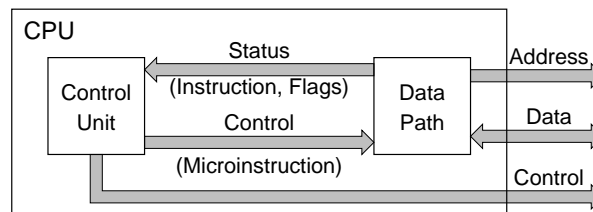


Figure 7.4: Interaction between the Data Path and the CPU Control Unit

In this section we show how to design a control unit that generates microinstructions in a timely fashion. Before the design of the control unit can begin, it is imperative to define

<sup>4</sup>Each  $\mu$ AL instruction is encoded as a single microinstruction.

the data path as well as the  $\mu$ AL routines for interpreting each of the machine language instructions. In implementing the control unit, there are two distinct aspects to deal with: (i) the proper sequencing through the *microroutines* (i.e., encoded form of  $\mu$ AL routines), and (ii) the generation of the appropriate microinstruction. The first step is to combine the routines for all of the machine language instructions into a single  $\mu$ AL program. Notice that the  $\mu$ AL routines given so far contained intra-routine sequencing information only. Combining the routines involves stitching them together. This job is similar to what the *link editor* or *linker* does when combining multiple ML object code files into a single ML executable program.

We shall use the now familiar data path of Figure 7.2 for designing the control unit. Furthermore, for clarity of presentation, we shall restrict ourselves to the following three instructions: (i) `addu rd, rs, rt`, (ii) `lw rt, offset(rs)`, and (iii) `beq rs, rt, offset`. That is, we will assume for the time being that this processor implements only these three instructions. The individual  $\mu$ AL routines for interpreting these instructions are available in Tables 7.2-7.4. The  $\mu$ AL routine for the instruction fetch part is the same for all three instructions. Table 7.5 combines the  $\mu$ AL routines in Tables 7.2-7.4 into a single  $\mu$ AL program starting at step 0.

In this  $\mu$ AL program, the routine for instruction fetch is written just once, and takes steps 0 through 3. In step 3, a new  $\mu$ AL operation—`goto step  $n$` —has been added to indicate to the control unit that a multi-way branch is required in the  $\mu$ AL program, based on the opcode of the decoded instruction. Notice also that a `goto step 0`  $\mu$ AL operation has been added to the last  $\mu$ AL instruction in each routine to indicate that control has to go back to step 0 after the execution of that  $\mu$ AL instruction. The `goto`  $\mu$ AL operations are for the benefit of the CPU control unit (to facilitate its sequencing function), and are not meant for the data path.

The CPU control unit essentially executes an infinite loop at the microarchitecture level. This unit can be designed as a finite state machine (FSM). In order to do this, we can develop a state transition diagram based on Table 7.5. Figure 7.5 presents a Moore-type state transition diagram for this FSM. In this diagram, each step of Table 7.5 is implemented by a state, which decides the  $\mu$ AL instruction to be generated when in that state. The figure also indicates the conditions that cause the control unit FSM to go from one state to another. Most of the transitions are not marked by a condition, which means that those transitions occur unconditionally. The first 4 states, `S0 - S3`, correspond to the instruction fetch routine; the  $\mu$ AL instruction to be performed in each of these states  $s$  corresponds to what is given in Table 7.5 for step  $s$ . At the end of the fetch  $\mu$ AL routine, the fetched ML instruction would have been decoded, and a multi-way branch occurs to the appropriate execute  $\mu$ AL routine. By specifying the states and their transitions, we specify the  $\mu$ AL instruction the CPU control unit must generate in order for the data path to fetch, decode, and execute every instruction in the ISA.

The state diagram implemented in this fashion has a separate routine for the execution part of each ML instruction. For an ISA that specifies hundreds of instructions, this ap-

Step No.	$\mu$ AL Instruction		Comments
	Next Step No. for Control Unit	$\mu$ AL Operations for Data Path	
<i>Fetch phase of every instruction</i>			
0	goto step $n$	PC $\rightarrow$ MAR; PC $\rightarrow$ AIR	<i>Decode; branch based on opcode</i>
1		M[MAR] $\rightarrow$ MDR; AIR + 4 $\rightarrow$ AOR	
2		MDR $\rightarrow$ IR	
3		AOR $\rightarrow$ PC	
<i>Execute phase of addu</i>			
4	goto step 0	R[rs] $\rightarrow$ AIR	<i>End of routine</i>
5		R[rt] + AIR $\rightarrow$ AOR	
6		AOR $\rightarrow$ R[rd]	
<i>Execute phase of lw</i>			
7	goto step 0	R[rs] $\rightarrow$ AIR	<i>End of routine</i>
8		SE(offset) + AIR $\rightarrow$ AOR	
9		AOR $\rightarrow$ MAR	
10		M[MAR] $\rightarrow$ MDR	
11		MDR $\rightarrow$ R[rt]	
<i>Execute phase of beq</i>			
12	if ( $\bar{Z}$ ) goto step 0	R[rs] $\rightarrow$ AIR	<i>Conditional end</i>
13		R[rt] == AIR $\rightarrow$ Z	
14		LS(SE(offset)) $\rightarrow$ AIR	
15		PC + AIR $\rightarrow$ AOR	
16	goto step 0	AOR $\rightarrow$ PC	<i>End of routine</i>

Table 7.5: A  $\mu$ AL Program for Interpreting three MIPS ML Instructions for Execution in the Data Path of Figure 7.2

proach is likely to produce an FSM with thousands of states. A number of states in such an FSM can be combined by considering that the execution routines of similar instructions have many common states. For instance, the first 3 states in the execution routine of a memory-referencing instruction deal with computing the effective address, and will be the same for all memory-referencing instructions. Just like we used a common fetch routine for all ML instructions, we can use a common address calculation routine for all memory-referencing instructions in the MIPS ISA. If we take this approach, we can get a reduced FSM as shown in Figure 7.6. This FSM will have far fewer states than the one given earlier.

Having developed a  $\mu$ AL program to interpret the three machine language instructions, the next step is to translate it into an equivalent microprogram, and to generate the microinstructions so as to accomplish the interpretation of instructions one after the other. Figure 7.7 gives a block diagram of a generic CPU control unit FSM. This FSM performs two functions:

- Sequencing through the microprogram

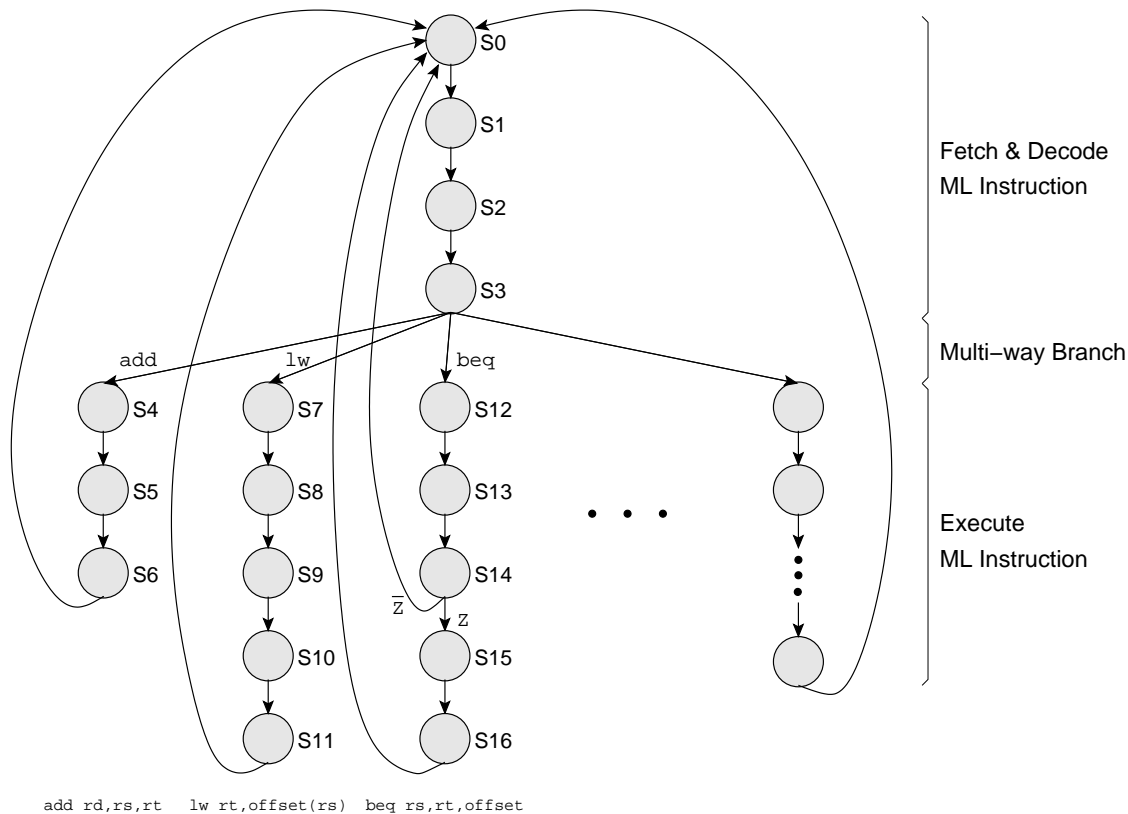


Figure 7.5: A State Diagram for a Control Unit for the MIPS Data Path of Figure 7.2

- Generating the microinstruction for the current state

The FSM *state* is stored in a **state register**. The contents of this register are updated by the combinational logic block called **next state generator**. Normally, the state value is incremented at each clock pulse, causing successive microinstructions to be generated. Every time a new ML instruction is decoded (which is in state 3 for our example), the corresponding microinstruction includes the binary equivalent of a **goto  $S_n$   $\mu$ AL** operation. This information is used by **next state generator** to generate a different state—the state corresponding to the beginning of the execute microroutine for that instruction—to be loaded into the **state register**. Thus, the control unit is able to generate the microinstructions for executing the newly decoded machine language instruction. While going through the execute microroutine, if the current microinstruction includes the binary equivalent of a **goto  $S_0$   $\mu$ AL** operation, **next state generator** resets the **state register** to zero, so that in the next cycle the CPU can begin fetching the next machine language instruction. Thus, the updation of the **state register** is based on (i) the current contents of the **state register**, (ii) the outputs of the instruction decoder, and (iii) the contents of the flags register.

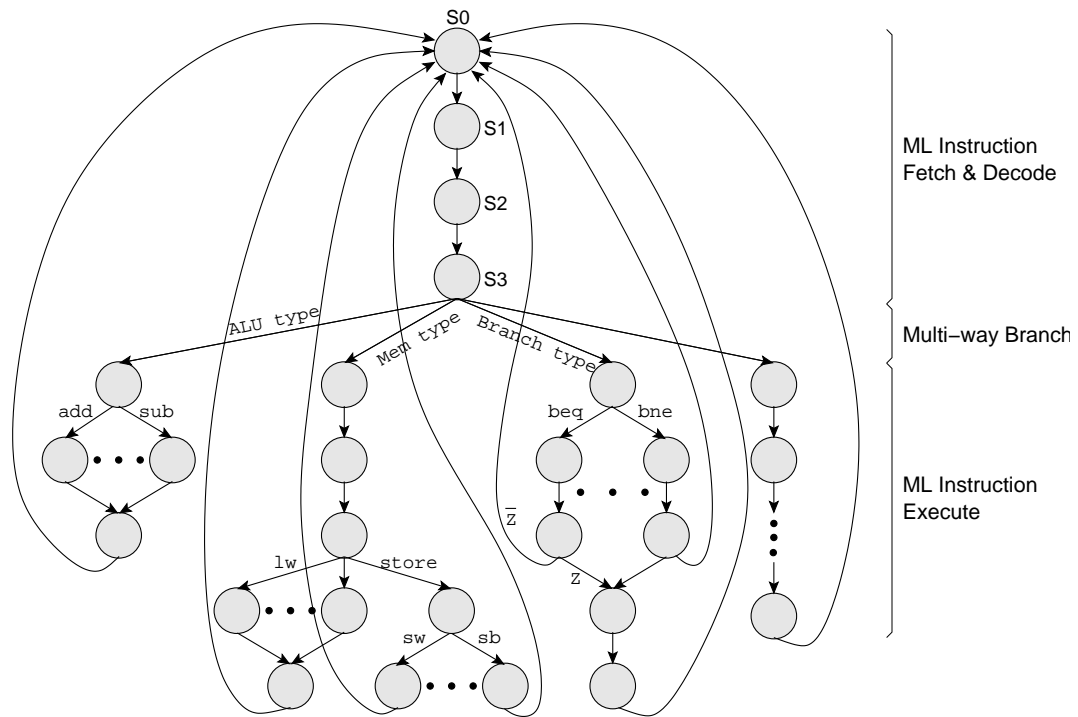


Figure 7.6: A Reduced State Diagram for a CPU Control Unit for the MIPS Data Path of Figure 7.2

The **microinstruction generation logic** takes the contents of **state** as input and generates the corresponding microinstruction, which can be optionally latched onto a  $\mu$ IR (microinstruction register). You must be careful not to confuse **state** with PC,  $\mu$ IR with IR, and microinstructions with machine language instructions.

At the digital logic level, the sequencing part of the control unit FSM can be implemented using a sequencer plus a decoder or one flip-flop per state. The sequencer can be built in many ways. One possibility is to use a counter or shift register with synchronous reset and parallel loading facility. The **microinstruction generation logic** can also be built in more than one way. Two common methods involve the use of either discrete logic or ROM. The control units so designed are called *hardwired control* and *microprogrammed control*, respectively.

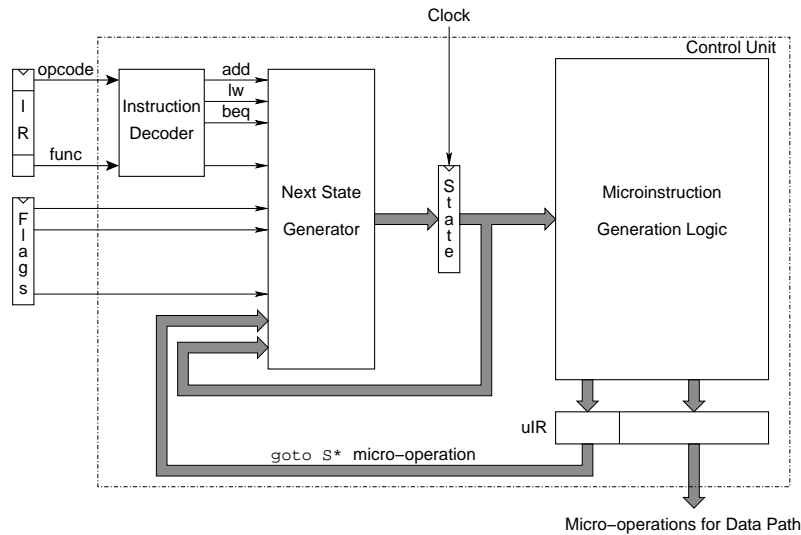


Figure 7.7: Block Diagram of a Generic CPU Control Unit

## 7.5 Defining CPU Data Path Interconnects for High Performance

The previous sections discussed the basics of designing a data path for interpreting machine language instructions. We used a very simple data path for ease of understanding. Although such a data path may not provide high performance, it may be quite sufficient for certain embedded applications. The data path of general-purpose computers is far more complex. In this section we will introduce high-performance versions of the CPU part of the data path.

The performance of a computer depends on a number of factors, many of which are related to the design of the CPU data path. Three of the most important factors are the strength of the machine language instructions, the number of clock cycles required to interpret a machine language instruction, and the clock speed. A powerful machine language instruction performs a complex operation and accomplishes more than what a simple instruction accomplishes, although it might take several additional clock cycles for execution. The strength of instructions is an issue that is dealt with at the ISA level, and does not come within the microarchitecture level.

Clock speed has a major influence on performance, and depends on the technology used to implement the electronic circuits. The use of densely packed, small transistors to fabricate the digital circuits leads to high clock speeds. Thus, implementing the entire CPU on a single VLSI chip allows much higher clock speeds than would be possible if several

chips were used. Clock speed is an issue that is dealt with at the digital logic level, and does not fall within the microarchitecture level.

The third factor in performance, namely the number of clock cycles required to interpret an instruction, is very much a microarchitectural issue. Speed improvements due to better microarchitectures, while less amazing than that due to faster circuits, have nevertheless been impressive. There are two ways to reduce the average number of cycles per instruction: (i) reduce the number of cycles needed to interpret each instruction, and (ii) overlap the interpretation of multiple instructions so that the average number of cycles per instruction is reduced. Both involve significant modifications to the CPU data path, primarily to add more connectivity and latches.

### 7.5.1 Multiple-Bus CPU Data Paths

In order to reduce the time required to interpret ML instructions, we need to redesign the data path so that several micro-operations can be performed in parallel. The more interconnections there are between the different blocks of a data path, the more the data transfers that can happen in parallel in a clock cycle. The performance of a data path does depend on the connectivity it provides. In a bus-based CPU data path, the number of buses provided is probably the most important limiting factor governing the number of cycles required to interpret machine language instructions, as most of the microinstructions would need to use the bus(es) in one way or other. A CPU data path with a single internal bus, such as the one given in Figure 7.2, will take several cycles to fetch the register operands and to write the ALU result to the destination register. This is because it takes one cycle to transfer each register operand to the ALU, and one cycle to transfer the result from the ALU to a register. An obvious way to reduce the number of cycles required to interpret machine language instructions is to include multiple buses in the data path, which makes it possible to parallelly transfer multiple register values to the ALU. Before going into specific multiple bus-based data paths, a word of caution is in order. Buses so take up significant chip area. Furthermore, they may need to cross at various points in the chip, which can make it difficult to do the layout at the VLSI level.

#### A 2-Bus Data Path

The single-bus CPU data path given in Figure 7.2 can be enhanced by adding one more internal bus. The additional bus can be connected to the blocks in different ways. Figure 7.8 shows one possible way of connecting the second bus. In the figure, the new bus is shown in darker shade, and is used primarily for routing the ALU result back to different registers. A unidirectional path is provided from the first bus to the second. Because the ALU result can be routed through a bus that is different from the one used for routing the source operand, there is no need to buffer the ALU result in AOR, and so we do not include AOR in this data path. Notice that to perform a register read and a register write in the

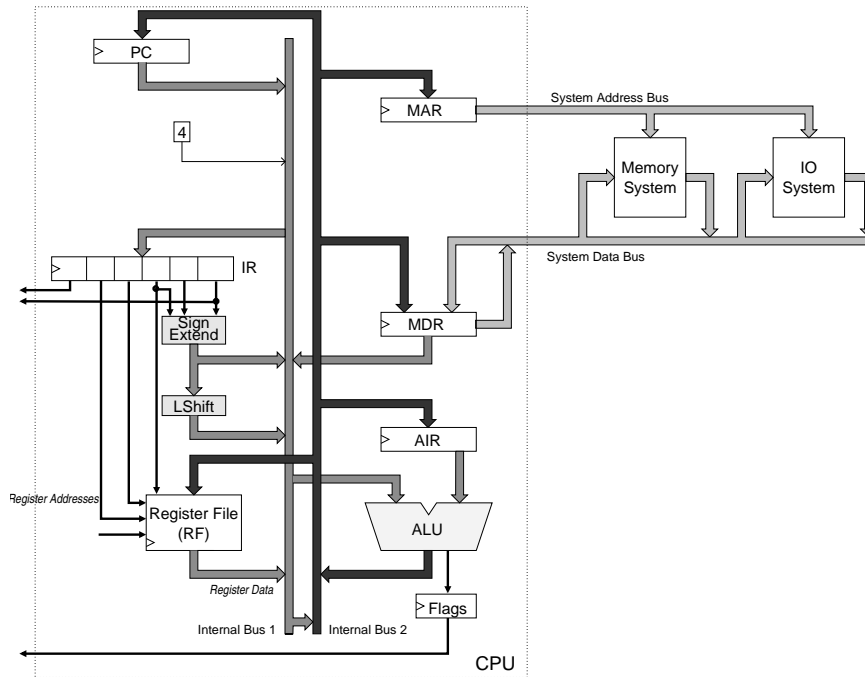


Figure 7.8: A 2-Bus Based CPU Data Path for Interpreting the MIPS ISA Subset

same clock cycle, the register file requires two ports (a read port and a write port).

Table 7.6 gives a  $\mu$ AL interpretation of `addu rd, rs, rt` in this 2-bus data path. On comparing this routine against that given in Table 7.2, we can see that steps 2 and 3 have been combined into a single step, as we can use the two bus to do both transfers in parallel. In the new step 3, when the instruction is being decoded, the `rs` field of `IR` is used to read GPR numbered `rs` into `AIR` in a *speculative* manner. That is, this transfer is done before completing instruction decoding, with the anticipation that it would be required later. Because most of the MIPS instructions specify `rs` as one of the source operands, this technique results in a saving of one clock cycle most of the time. Notice that the addition  $\mu$ AL instruction (step 4) directly writes the result into GPR `rd`, and therefore the number of clock cycles required to interpret the instruction is reduced by two compared to that given in Table 7.2.

Table 7.7 gives a  $\mu$ AL interpretation of `lw rt, offset(rs)` in this 2-bus data path. This  $\mu$ AL routine has 7 steps, which is 2 steps fewer than the one given in Table 7.3 for the single bus data path.

Step No.	$\mu$ AL Instruction	
	for Data Path	for Control Unit
<i>Fetch and Decode phase</i>		
0	PC $\rightarrow$ MAR;	PC $\rightarrow$ AIR
1	M[MAR] $\rightarrow$ MDR;	AIR + 4 $\rightarrow$ AOR
2	MDR $\rightarrow$ IR;	AOR $\rightarrow$ PC
3	R[rs] $\rightarrow$ AIR;	Decode instr
<i>Execute phase</i>		
4	R[rt] + AIR $\rightarrow$ R[rd]	

Table 7.6: A  $\mu$ AL Routine for Interpreting the MIPS ML Instruction Represented Symbolically as `addu rd, rs, rt`, in the 2-Bus Data Path

Step No.	$\mu$ AL Instruction	
	for Data Path	for Control Unit
<i>Fetch and Decode phase</i>		
0	PC $\rightarrow$ MAR;	PC $\rightarrow$ AIR
1	M[MAR] $\rightarrow$ MDR;	AIR + 4 $\rightarrow$ AOR
2	MDR $\rightarrow$ IR;	AOR $\rightarrow$ PC
3	R[rs] $\rightarrow$ AIR;	Decode instr
<i>Execute phase</i>		
4	SE(offset) + AIR $\rightarrow$ MAR	
5	M[MAR] $\rightarrow$ MDR	
6	MDR $\rightarrow$ R[rt]	

Table 7.7: A  $\mu$ AL Routine for Interpreting the MIPS ML Instruction Represented Symbolically as `lw rt, offset(rs)`, in the 2-Bus Data Path

## 7.5.2 Direct Path-based CPU Data Path

As we keep adding more buses to the CPU data path, we eventually get a data path that has many point-to-point connections or direct paths. If we are willing to use a large number of direct path connections, it is better to redesign the CPU data path.

Figure 7.9 presents a direct path-based data path for the MIPS ISA subset. In this data path, point-to-point connections or direct paths are provided between each pair of components that transfer data, instead of providing one or more buses. This approach allows many register transfers to take place simultaneously, thereby permitting multiple  $\mu$ AL operations to be executed in a clock cycle. This probably makes the data path's functioning easier to visualize, because each interconnection is used for a specific transfer as opposed to the situation in a bus-based data path where a bus transaction changes from clock cycle to clock cycle. Notice, however, that a direct path based data path will be more

tailored to a particular ISA.

If multiple sources feed a block, a multiplexer is used to select the required source in each cycle. For instance, in our data path, the second input of the ALU can receive data from register `Rrt` as well as from register `Offset`. Therefore, we use a multiplexer to select one of these inputs. Another modification is that there are two paths emanating from the register file. If both paths are to be active in the same cycle, then the register file needs to have two *read ports*.

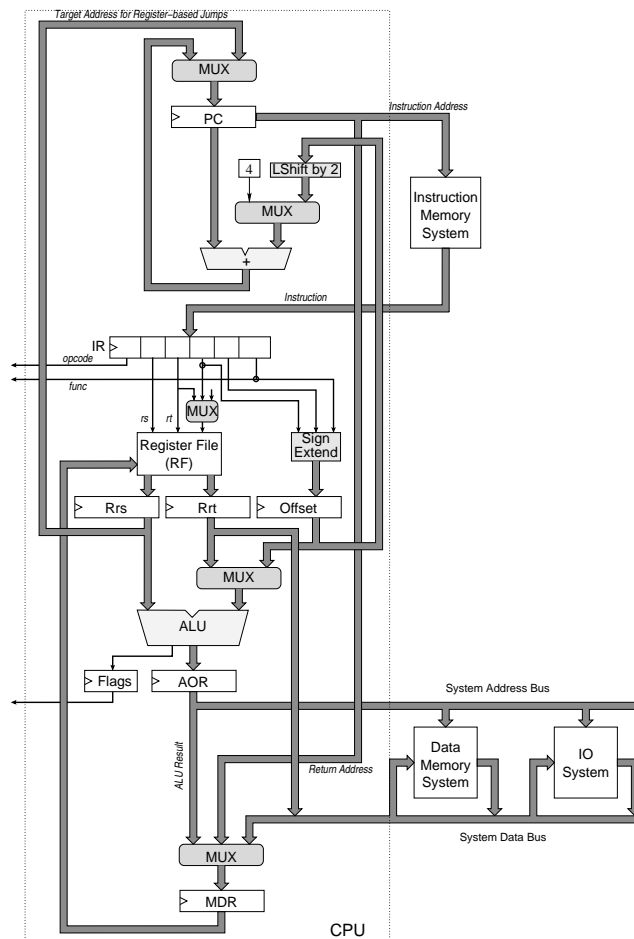


Figure 7.9: A Direct Path-Based CPU Data Path for the MIPS ISA Subset

Table 7.8 gives a  $\mu$ AL interpretation of `lw rt, offset(rs)` in this direct path-based data path. Notice that this  $\mu$ AL routine requires only 5 clock cycles to complete, which is 2 cycles less than the one given in Table 7.7 for the 2-bus data path.

Step No.	$\mu$ AL Instruction	
	for Data Path	for Control Unit
<i>Fetch and Decode phase</i>		
0	IM[PC] $\rightarrow$ IR; PC + 4 $\rightarrow$ PC	
1	R[rs] $\rightarrow$ Rrs; R[rt] $\rightarrow$ Rrt; SE(offset) $\rightarrow$ Offset	Decode instr
<i>Execute phase</i>		
2	Rrs + Offset $\rightarrow$ AOR	
3	DM[AOR] $\rightarrow$ MDR	
4	MDR $\rightarrow$ R[rt]	

Table 7.8: A  $\mu$ AL Routine for Interpreting the MIPS ML Instruction Represented Symbolically as `lw rt, offset(rs)`, in the Direct Path based Data Path

## 7.6 Pipelined Data Path: Overlapping the Interpretation of Multiple Instructions

The use of multiple buses and direct paths in the CPU data path enabled us to reduce the number of cycles required to interpret machine language instructions. Can we reduce the number of cycles further by increasing the connectivity? It turns out that this is difficult, because the remaining steps are somewhat sequential in nature. However, instead of reducing the number of cycles required to interpret each instruction, we can reduce the total number of cycles required to interpret a sequence of instructions by overlapping the interpretation of multiple instructions. In order to overlap the execution of multiple machine language instructions, a commonly used technique is *pipelining*, similar to the assembly-line processing used in factories.

In the data paths that we studied so far, after a machine language instruction is fetched and passed onto the instruction decoder, the fetch part of the CPU sits idle until the interpretation of the instruction is completed. By contrast, in a pipelined data path, the fetch part of the data path utilizes this time to fetch the next instruction. Similarly, after the instruction decoder has decoded an instruction, it starts decoding the next instruction, without waiting for the previous instruction's execution to be completed.

It is important to understand that pipelining does not result in individual instructions being executed faster; rather, it increases the rate at which instructions are interpreted. With a four-stage pipeline, the rate at which instructions are completed is up to four times that of an unpipelined processor. Based on this discussion, the performance obtained with pipelining appears to be proportional to the number of pipeline stages. Unfortunately, this is not the case. For a variety of reasons, one of the pipeline stages may not be able to perform its function for a given instruction in a particular clock cycle.

### 7.6.1 Defining a Pipelined Data Path

To begin with, we need to determine what should happen in the data path during every clock cycle, and make sure that the same hardware resource is not used by two different operations during the same clock cycle. This makes it difficult to use a bus-based data path, as the bus is a heavily used resource. Therefore, our starting point is the direct path-based data path that we saw in Figure 7.9. In order to avoid conflicts for some of the hardware blocks, we need to replicate some of the blocks in the direct path-based data path. For example, a single PC cannot store the addresses of two different ML instructions at the same time. Therefore, the pipelined data path must provide multiple copies of PC, one for each instruction that is being interpreted in the CPU.

An example pipelined data path for the MIPS subset is given in Figure 7.10. In this data path, an ML instruction uses the major hardware blocks in different clock cycles, and hence overlapping the interpretation of multiple instructions introduces relatively fewer conflicts.

Pipelining has been used since the 1960s to increase instruction throughput. Since then, a number of hardware and software features have been developed to enhance the effectiveness and efficiency of pipelining. Not all pipelined data paths have all of these characteristics; the ones we discuss below are quite prevalent.

**Dual-ported Memory:** If instructions are to be introduced into the pipelined data path at a rate of one per clock cycle, then no two instructions must attempt to use the same hardware resource in the same clock cycle. In order to achieve this, no instruction must be using the same hardware resource in two different pipeline stages. This condition is violated by memory referencing instructions, because they access the memory for fetching the instruction during the fetch stage, and access the memory again, later in the memory stage, for accessing data. Two separate memories are needed then, or at least the appearance of two separate memories, so that instructions and data can be accessed simultaneously.

**Multi-ported Register File:** If any of the instructions need to read from (or write into) the register file in multiple cycles, then multiple read (write) ports need to be provided.

**Interconnect:** Pipelined data paths typically use the direct path-based approach discussed earlier, which uses direct connections between registers. A bus-based data path is not suitable for pipelining, because it prevents more than one instruction from using the bus in the same clock cycle. The direct path approach allows many register transfers to take place simultaneously in each pipeline stage, which indeed they must if the goal of allowing many activities to proceed simultaneously is to be achieved. In fact, we may have to add more interconnections to allow for more parallel data transfers.

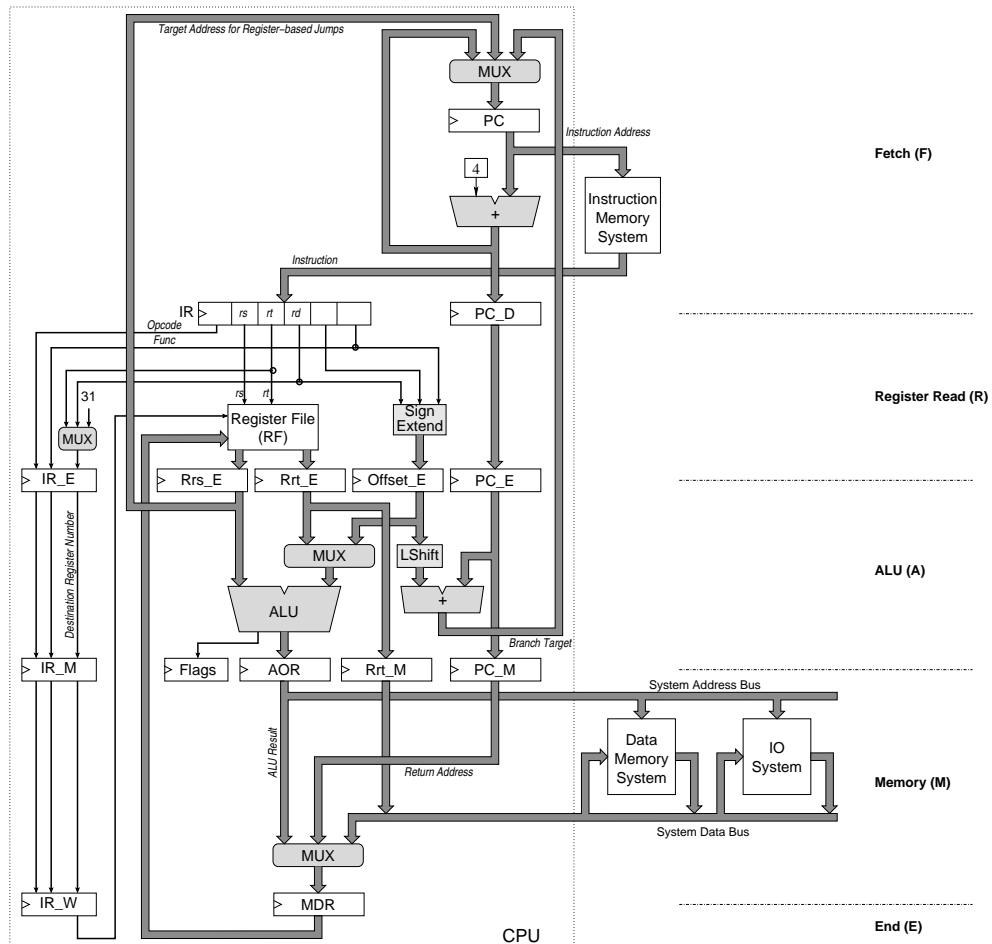


Figure 7.10: A Pipelined CPU Data Path for the MIPS ISA Subset

**Pipeline Registers or Latches:** When an instruction moves from one pipeline stage to the next, the vacated stage is occupied by the next instruction. Because of this, whatever an instruction needs to know about itself at each pipeline stage must be carried along until that stage is reached. Such information might include opcode, data values, etc. Each instruction must “carry its own bags,” so to speak. Thus, portions of both the data and the instruction must travel from stage to stage, even if they are not modified by a particular stage. To do this in a convenient fashion, extra registers, called *pipeline latches*, are typically used in between the pipeline stages. For example, part of the IR contents are included in the pipeline registers IR\_E and IR\_M.

**Additional Hardware Resources:** Pipelined data paths usually need additional hardware resources to avoid resource conflicts between simultaneously interpreted instructions. A likely candidate is a separate incremter to increment the PC, freeing the ALU for doing arithmetic/logical operations in every clock cycle.

### 7.6.2 Interpreting ML Instructions in a Pipelined Data Path

The objective of designing a pipelined data path is to permit the interpretation of multiple ML instructions at the same time, in a pipelined manner. The  $\mu$ AL routines used for interpreting each ML instruction is similar to those for the direct path-based data path. Table 7.9 illustrates how the  $\mu$ AL routines for multiple ML instructions are executed in parallel.

Cycle No.	$\mu$ AL Instruction		
	lw rt, offset(rs)	addu rd, rs, rt	beq rs, rt, offset
0	$\mu$ AL Instruction for Fetch		
1	$\mu$ AL Instruction for Decode		
2	Rrs_E + Offset_E $\rightarrow$ AOR	$\mu$ AL Instruction for Decode	$\mu$ AL Instruction for Fetch
3	DM[AOR] $\rightarrow$ MDR	Rrs_E + Rrt_E $\rightarrow$ AOR	$\mu$ AL Instruction for Decode
4	MDR $\rightarrow$ R[r_W]	AOR $\rightarrow$ MDR	if (Rrs_E == Rrt_E) PC_E + LS(Offset_E) $\rightarrow$ PC
5		MDR $\rightarrow$ R[r_W]	
6			

Table 7.9: Overlapped Execution of  $\mu$ AL Routines for Interpreting Three MIPS ML Instructions in the Pipelined Data Path of Figure 7.10

### 7.6.3 Control Unit for a Pipelined Data Path

Converting an unpipelined data path into a pipelined one involves several modifications, as we just saw. Next, let us look at the design of control units for pipelined data paths. With a pipelined data path, in each clock cycle, the control unit has to generate appropriate  $\mu$ AL instructions for interpreting the ML instructions present in each pipeline stage during that cycle. This calls for some major changes in the control unit, perhaps even more than what was required for the data path. A simple approach is to generate at instruction decode time all of the  $\mu$ AL instructions required to interpret that instruction in the subsequent clock cycles. (in different pipeline stages). These  $\mu$ AL instructions are passed on to the pipelined data path, which carries them along with the instruction by extending the pipeline registers. Then, in each pipeline stage, the appropriate  $\mu$ AL instruction is executed by pulling it out of its pipeline register.

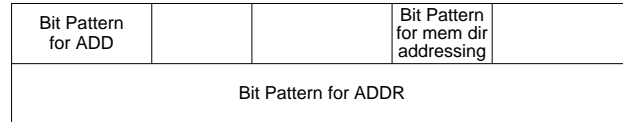
An alternate approach is for each pipeline stage to have its own control unit, so to speak.

Depending on the opcode of the instruction that is currently occupying pipeline stage  $i$ , the  $i^{\text{th}}$  control unit generates the appropriate  $\mu\text{AL}$  instruction and passes it to stage  $i$ .

Intel's Pentium processor used a relatively short 5-stage pipeline as in the MIPS R2000 and R3000 processors. The Pentium Pro/II/III series of processors used a deeper pipeline having 12 stages, whereas the recent Pentium IV uses a very deep pipeline having 20 stages.

### Exercises

1. Consider a non-MIPS ISA that has variable length instructions. One of the instructions in this ISA is ADD (ADDR), which means add the contents of memory location ADDR to ACC register. This instruction has the following encoding; notice that it occupies two memory locations.



Specify a  $\mu$ AL routine to carry out the interpretation of this instruction in the data path given in Figure 7.11.

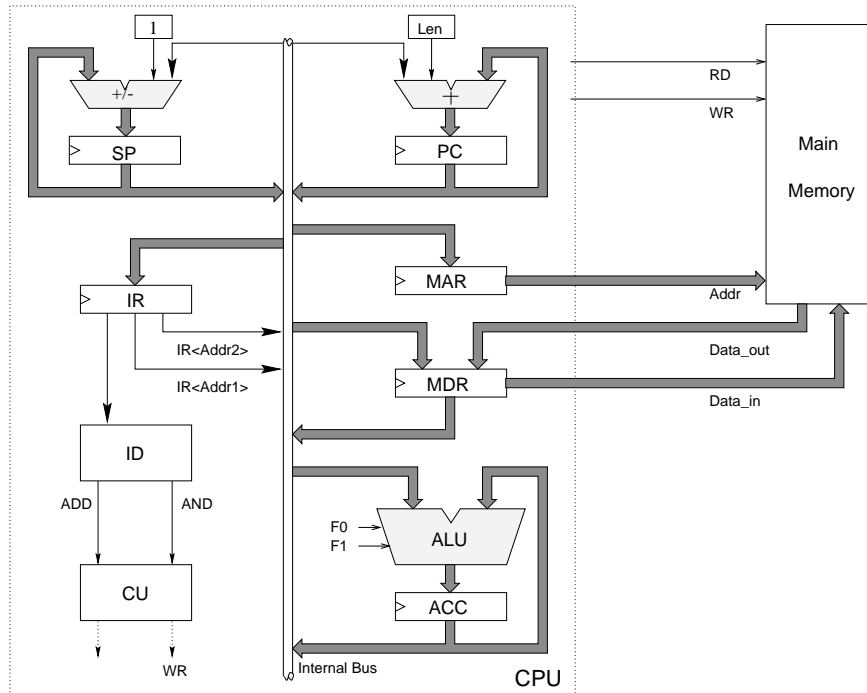


Figure 7.11: A Single Bus based CPU Data Path for Interpreting an ACC based ISA

2. Consider the non-MIPS instruction ADD (ADDR1), (ADDR2), where ADDR1 and ADDR2 are memory locations whose contents needed to be added. The result is to be stored in memory location ADDR1. This instruction has the following encoding; notice that

addresses ADDR1 and ADDR2 are specified in the words subsequent to the word that stores the ADD opcode.

Bit Pattern for ADD	Bit Pattern for mem dir addressing		Bit Pattern for mem dir addressing	
Bit Pattern for ADDR1				
Bit Pattern for ADDR2				

Specify the data transfer operations required to fetch and execute this instruction in the data path given in Figure 7.11. Notice that register ACC is a part of the ISA, and therefore needs to retain the value that it had before interpreting the current instruction. You are allowed to grow the stack in the direction of lower memory addresses.