Chapter 3 Processor Implementation
(Revision number 20)

The previous chapter dealt with issues involved in deciding the instruction-set architecture of the processor. This chapter deals with the implementation of the processor once we have decided on an instruction set. The instruction-set is not a description of the implementation of the processor. It serves as a contract between hardware and software. For example, once the instruction-set is decided, a compiler writer can generate code for different high-level languages to execute on a processor that implements this contract. Naturally, we can we have different implementations of the same instruction set. As we will see in this chapter, a number of factors go into deciding the implementation choice that one may make.

3.1 Architecture versus Implementation

First, let us understand why this distinction between architecture and implementation is important.

1. Depending on cost/performance, several implementations of the same architecture may be possible and even necessary to meet market demand. For example, it may be necessary to have a high performance version of the processor for a server market (such as a web server); at the same time, there may be a need for lower performance version of the same processor for an embedded application (such as a printer). This is why we see a *family* of processors adhering to a particular architecture description, some of which may even be unveiled by a vendor simultaneously (e.g., Intel Xeon series, IBM 360 series, DEC PDP 11 series, etc.).

2. Another important reason for decoupling architecture from implementation is to allow parallel development of system software and hardware. For example, it becomes feasible to validate system software (such as compilers, debuggers, and operating systems) for a new architecture even prior to an implementation of the architecture is available. This cuts down the time to market a computer system drastically.

3. Customers of high-performance servers make a huge investment in software. For example, Oracle database is a huge and complex database system. Such software systems evolve more slowly compared to generations of processors. Intel co-founder Gordon Moore predicted in 1965 that the number of transistors on a chip would double every two years. In reality, the pace of technology evolution has been even faster, with processor speed doubling every 18 months. This means that a faster processor can hit the market every 18 months. You would have observed this phenomenon if you have been following the published speeds of processors appearing in the market year after year. Software changes more slowly compared to hardware technology; therefore, it is important that *legacy* software run on new releases of processors. This suggests that we want to maintain the contract (i.e., the instruction-set) the same so that much of the software base (such as compilers and related tool

sets, as well as fine-tuned applications) remain largely unchanged from one generation of processor to the next. Decoupling architecture from implementation allows this flexibility to maintain binary compatibility for legacy software.

3.2 What is involved in Processor Implementation?

There are several factors to consider in implementing a processor: price, performance, power consumption, cooling considerations, operating environment, etc. For example, a processor for military applications may require a more rugged implementation capable of withstanding harsh environmental conditions. The same processor inside a laptop may not require as rugged an implementation.

There are primarily two aspects to processor implementations:

1. The first aspect concerns the organization of the electrical components (ALUs, buses, registers, etc.) commensurate with the expected price/performance characteristic of the processor.
2. The second aspect concerns thermal and mechanical issues including cooling and physical geometry for placement of the processor in a printed circuit board (often referred to as *motherboard*).

The above two issues are specifically for a single-chip processor. Of course, the hardware that is "inside a box" is much more than a processor. There is a host of other issues to consider for the box as a whole including printed circuit boards, backplanes, connectors, chassis design, etc. In general, computer system design is a tradeoff along several axes. If we consider just the high-end markets (supercomputers, servers, and desktops) then the tradeoff is mostly one of *price/performance*. However, in embedded devices such as cell phones, a combination of three dimensions, namely, *power* consumption, *performance*, and *area* (i.e., size), commonly referred to as *PPA*, has been the primary guiding principle for design decisions.

Super Computers	Servers	Desktops & Personal Computers	Embedded
High performance primary objective	***Intermediate performance and cost***	***Low cost primary objective***	***Small size, performance, and low power consumption primary objectives***

Computer design is principally an empirical process, riddled with tradeoffs along the various dimensions as mentioned above.

In this chapter, we will focus on processor implementation, in particular the design of the datapath and control for a processor. The design presented in this chapter is a basic one. In Chapter 5, we explore pipelined processor implementation.

We will first review some key hardware concepts that are usually covered in a first course on logic design.

3.3 Key hardware concepts

3.3.1 Circuits

Combinational logic: The output of such a circuit is a Boolean combination of the inputs to the circuit. That is, there is no concept of a *state* (i.e., memory). Basic logic gates (AND, OR, NOT, NOR, NAND) are the building blocks for realizing such circuits. Another way of thinking about such circuits is that there is no *feedback* from output of the circuit to the input.

Consider a patch panel that mixes different microphone inputs and produces a composite output to send to the speaker. The output of the speaker depends on the microphones selected by the patch panel to produce the composite sound. The patch panel is an example of a combinational logic circuit. Examples of combinational login circuits found in the datapath of a processor include multiplexers, de-multiplexers, encoders, decoders, ALU's.

Sequential logic: The output of such a circuit is a Boolean combination of the current inputs to the circuit and the current *state* of the circuit. A memory element, called a *flip-flop*, is a key building block of such a circuit in addition to the basic logic gates that make up a combinational logic circuit.

Consider a garage door opener's control circuit. The "input" to the circuit is the clicker with a single push-button and some switches that indicate if the door is all the way up or all the way down. The "output" is a signal that tells the motor to raise or lower the door. The direction of motion depends on the "current state" of the door. Thus, the garage door opener's control circuit is a sequential logic circuit. Examples of sequential logic circuits usually found in the datapath of a processor include registers and memory.

3.3.2 Hardware resources of the datapath

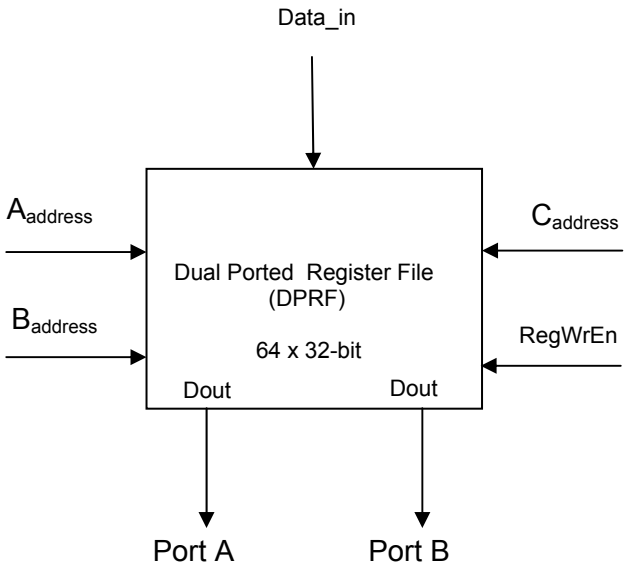
The datapath of a processor consists of combinational and sequential logic elements. With respect to the instruction-set of LC-2200 that we summarized in Chapter 2, let us identify the datapath resources we will need.

We need **MEMORY** to store the instructions and operands. We need an **Arithmetic Logic Unit** (**ALU**) to do the arithmetic and logic instructions. We need a **REGISTER-FILE** since they are the focal point of action in most instruction-set architectures. Most instructions use them. We need a **Program Counter** (henceforth referred to simply as **PC**) in the datapath to point to the current instruction and for implementing the branch and jump instructions we discussed in Chapter 2. When an instruction is brought from memory it has to be kept somewhere in the datapath; so we will introduce an **Instruction Register** (**IR**) to hold the instruction.

As the name suggests, a register-file is a collection of architectural registers that are visible to the programmer. We need control and data lines to manipulate the register file. These include address lines to name a specific register from that collection, and data lines for reading from or writing into that register. A register-file that allows a single register to be read at a time is called a *single ported register file (SPRF)*. A register-file that allows two registers to be read simultaneously is referred to as a *dual ported register file (DPRF)*. The following example sheds light on all the control and data lines needed for a register-file.

Example 1:

Shown below is a dual-ported register file (DPRF) containing 64 registers. Each register has 32 bits. $A_{address}$ and $B_{address}$ are the register addresses for reading the 32-bit register contents on to Ports A and B, respectively. $C_{address}$ is the register address for writing Data_in into a chosen register in the register file. RegWrEn is the write enable control for writing into the register file. Fill in the blanks.



Answer:

(a) Data_in has _____**32**_____ wires

(b) Port A has _____**32**_____ wires

(c) Port B has _____**32**_____ wires

(d) $A_{address}$ has _____**6**_____ wires

(e) $B_{address}$ has _____**6**_____ wires

(f) $C_{address}$ has _____**6**_____ wires

(g) RegWrEn has _____**1**_____ wires

3.3.3 Edge Triggered Logic

The contents of a register changes from its current *state* to its new *state* in response to a clock signal.

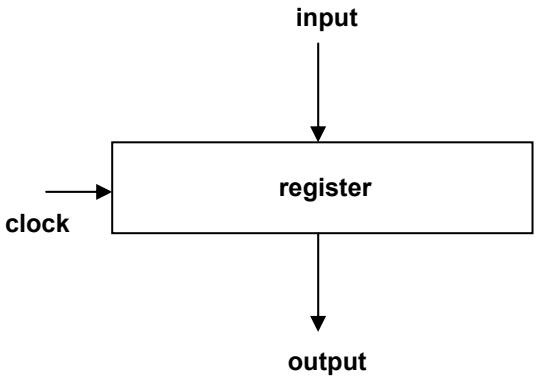


Figure 3.1: Register

The precise moment when the output changes state with respect to a change in the input depends on whether a storage element works on *level logic*[1] or *edge-triggered logic.* With level logic, the change happens as long as the clock signal is *high*. With edge-triggered logic, the change happens only on the *rising* or the *falling* edge of the clock. If the state change happens on the rising edge, we refer to it as *positive-edge-triggered logic;* if the change happens on the falling edge, we refer to it as *negative-edge-triggered logic.*

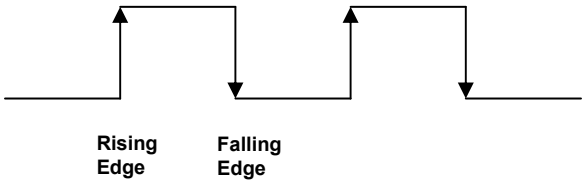
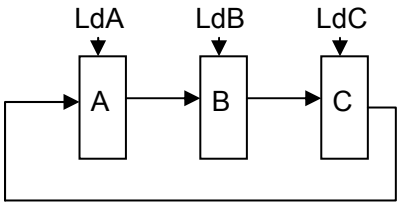


Figure 3.2: Clock

From now on in our discussion, we will assume positive edge triggered logic. We will discuss the details of choosing the width of the clock cycle in Section 3.4.2.

[1] It is customary to refer to a storage device that works with level logic as a *latch*. Registers usually denote edge-triggered storage elements.

Example 2:
Given the following circuit with three registers A, B, and C connected as shown:



Assume initially
A = 98
B = 765
C = 432

LdA, LdB, and LdC are the clock signals for registers A, B, and C, respectively. In the above circuit if LdA, LdB, and LdC are enabled in a given clock cycle.
What are the contents of A, B, and C in the next clock cycle?

Answer:
Whatever is at the input of the registers will appear at their respective outputs.

Therefore,

A = 432; B = 98; C = 765

The **MEMORY** element is special. As we saw in the organization of the computer system in Chapter 1, the memory subsystem is in fact completely separate from the processor. However, for the sake of simplicity in developing the basic concepts in processor implementation, we will include memory in the datapath design. For the purposes of this discussion, we will say that memory is not edge-triggered.

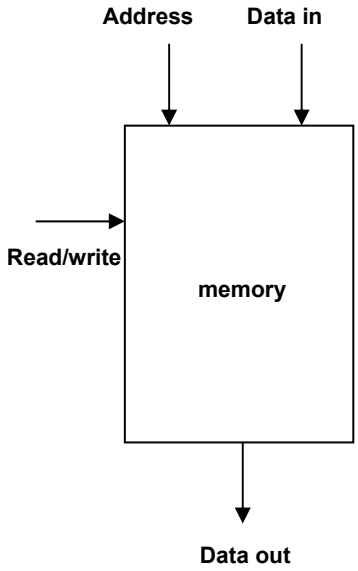


Figure 3.3: Memory

For example, to read a particular memory location, you supply the "Address" and the "Read" signal to the memory; after a finite amount of time (called the read access time of the memory) the contents of the particular address is available on the "Data out" lines.

Similarly, to write to a particular memory location, you supply the "Address", "Data in", and the "Write" signal to the memory; after a finite amount of time (the write access time) the particular memory location would have the value supplied via "Data in". We will deal with memory systems in much more detail in Chapter 9.

3.3.4 Connecting the datapath elements

Let us consider what needs to happen to execute an ADD instruction of LC-2200 and from that derive how the datapath elements ought to be interconnected.

1. **Step 1**: We need to use the PC to specify to the memory what location contains the instruction (Figure 3.4).

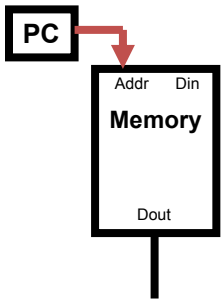


Figure 3.4: Step 1 – PC supplies instruction address to memory

2. **Step 2**: Once the instruction is read from the memory then it has to be stored in the IR (Figure 3.5).

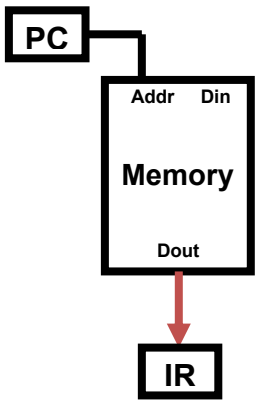


Figure 3.5: Step 2 – Instruction is read out of memory and clocked into IR

3. **Step 3**: Once the instruction is available in IR, then we can use the register numbers specified in the instruction (contained in IR) to read the appropriate registers from the REGISTER-FILE (dual ported similar to the one in Example 1), perform the addition using the ALU, and write the appropriate register into the REGISTER-FILE (Figure 3.6).

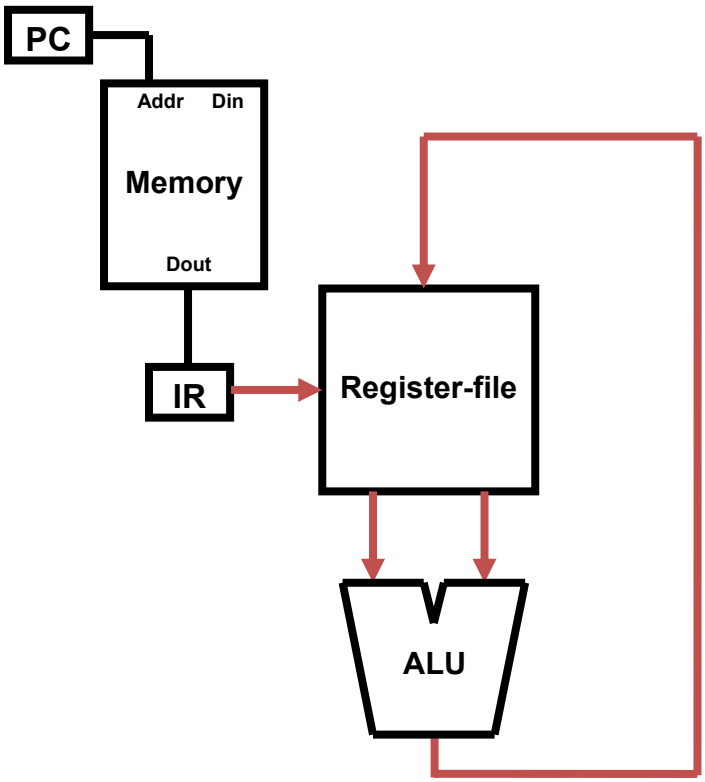


Figure 3.6: Step 3 – perform ADD two register values and store in third register

The above steps show the roadmap of the ADD instruction execution.

Let us see if all of the above steps can be completed in one clock cycle. As mentioned earlier, all the storage elements (except memory) are positive-edge triggered. What this means is that in one clock cycle we can transfer information from one storage element to another (going through combinational logic elements and/or memory) so long as the clock signal is long enough to account for all the intervening latencies of the logic elements. So for instance, Steps 1 and 2 can be completed in one clock cycle; but step 3 cannot be completed in the same clock cycle. For step 3, we have to supply the register number bits from IR to the register-file. However, due to the edge-triggered nature of IR, the instruction is available in IR only at the beginning of the **next** clock cycle (see Figure 3.7).

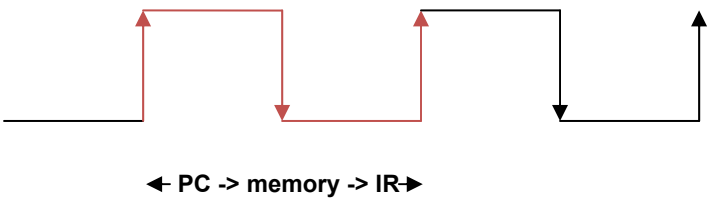


Figure 3.7: Steps 1 and 2 (first clock cycle)

It turns out that step 3 can be done in one clock cycle. At the beginning of the next clock cycle, the output of IR can be used to index the specific source registers needed for reading out of the register-file. The registers are read (similar to how memory is read given an address in the same cycle), passed to the ALU, the ADD operation is performed, and the results are written into the destination register (pointed to by IR again). Figure 3.8 illustrates the completion of step 3 in the second clock cycle.

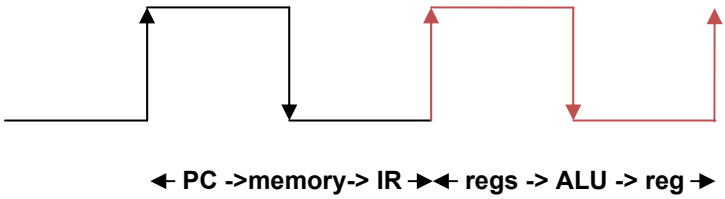


Figure 3.8: Step 3 (second clock cycle)

Determining the clock cycle time: Let us re-examine steps 1 and 2, which we said could be completed in one clock cycle. How wide should the clock cycle be to accomplish these steps? With reference to Figure 3.7, from the first rising edge of the clock, we can enumerate all the combinational delays encountered to accomplish steps 1 and 2:

- time that has to elapse for the output of PC to be stable for reading ($D_{r\text{-}output\text{-}stable}$);
- wire delay for the address to propagate from the output of PC to the Addr input of the memory ($D_{wire\text{-}PC\text{-}Addr}$);
- access time of the memory to read the addressed location ($D_{mem\text{-}read}$);
- wire delay for the value read from the memory to propagate to the input of IR ($D_{wire\text{-}Dout\text{-}IR}$);
- time that has to elapse for the input of IR to be stable before it the second rising edge in Figure 3.7, usually called the setup time ($D_{r\text{-}setup}$);
- time that the input of IR has to be stable after the second rising edge, usually called the hold time ($D_{r\text{-}hold}$).

The width of the clock for accomplishing steps 1-2 should be greater than the sum of all the above delays:

Clock width > $D_{r\text{-}ouput\text{-}stable} + D_{wire\text{-}PC\text{-}Addr} + D_{mem\text{-}read} + D_{wire\text{-}Dout\text{-}IR} + D_{r\text{-}setup} + D_{r\text{-}hold}$

We do such an analysis for each of the potential paths of signal propagation in every clock cycle. Thereafter, we choose the clock width to be greater than the worse *case delay for signal propagation* in the entire datapath. We will formally define the terms involved in computing the clock cycle time in Section 3.4.2.

Example 3:

Given the following parameters (all in picoseconds), determine the minimum clock width of the system. Consider only steps 1-3 of the datapath actions.

$D_{r\text{-}ouput\text{-}stable}$	(PC output stable)	-	20 ps
$D_{wire\text{-}PC\text{-}Addr}$	(wire delay from PC to Addr of Memory)	-	250 ps
$D_{mem\text{-}read}$	(Memory read)	-	1500 ps
$D_{wire\text{-}Dout\text{-}IR}$	(wire delay from Dout of Memory to IR)	-	250 ps
$D_{r\text{-}setup}$	(setup time for IR)	-	20 ps
$D_{r\text{-}hold}$	(hold time for IR)	-	20 ps
$D_{wire\text{-}IR\text{-}regfile}$	(wire delay from IR to Register file)	-	250 ps
$D_{regfile\text{-}read}$	(Register file read)	-	500 ps
$D_{wire\text{-}regfile\text{-}ALU}$	(wire delay from Register file to input of ALU)	-	250 ps
$D_{ALU\text{-}OP}$	(time to perform ALU operation)	-	100 ps
$D_{wire\text{-}ALU\text{-}regfile}$	(wire delay from ALU output to Register file)	-	250 ps
$D_{regfile\text{-}write}$	(time for writing into a Register file)	-	500 ps

Answer:

Steps 1 and 2 are carried out in one clock cycle.
Clock width needed for steps 1 and 2

$$C_{1\text{-}2} > D_{r\text{-}ouput\text{-}stable} + D_{wire\text{-}PC\text{-}Addr} + D_{mem\text{-}read} + D_{wire\text{-}Dout\text{-}IR} + D_{r\text{-}setup} + D_{r\text{-}hold}$$
$$> 2060 \text{ ps}$$

Step 3 is carried out in one clock cycle.
Clock width needed for step 3

$$C_3 > D_{wire\text{-}IR\text{-}regfile} + D_{regfile\text{-}read} + D_{wire\text{-}regfile\text{-}ALU} + D_{ALU\text{-}OP} + D_{wire\text{-}ALU\text{-}regfile} + D_{regfile\text{-}read}$$
$$> 1850 \text{ ps}$$

Minimum clock width > worse case signal propagation delay
$> MAX(C_{1\text{-}2}, C_3)$
> 2060 ps

The above parameters are fairly accurate circa 2007. What should be striking about these numbers is the fact that wire delays dominate.

3.3.5 Towards bus-based Design

We made up ad hoc connections among the datapath elements to get this one instruction executed. To implement another instruction (e.g., LD), we may have to create a path from the memory to the register file. If we extrapolate this line of thought, we can envision every datapath element connected to every other one. As it turns out, this is neither necessary nor the right approach. Let us examine what is involved in connecting the ALU to the register file. We have to run as many wires as the width of the datapath between the two elements. For a 32-bit machine, this means 32 wires. You can see wires

quickly multiply as we increase the connectivity among the datapath elements. Wires are expensive in terms of taking up space on silicon and we want to reduce the number of wires so that we can use the silicon real estate for active datapath elements. Further, just because we have more wires we do not necessarily improve the performance. For example, the fact that there are wires from the memory to the register file does not help the implementation of ADD instruction in any way.

Therefore, it is clear that we have to think through the issue of connecting datapath elements more carefully. In particular, the above discussion suggests that perhaps instead of dedicating a set of wires between every pair of datapath elements, we should think of designing the datapath sharing the wires among the datapath elements. Let us investigate how many sets of wires we need and how we can share them among the datapath elements.

Single bus design: One extreme is to have a single set of wires and have all the datapath elements share this single set of wires. This is analogous to what happens in a group meeting: one person talks and the others listen. Everyone takes a turn to talk during the meeting if he/she has something to contribute to the discussion. If everyone talks at the same time, there will be chaos of course. This is exactly how a *single bus system* – a single set of wires shared by all the datapath elements – works. Figure 3.9 shows such a system.

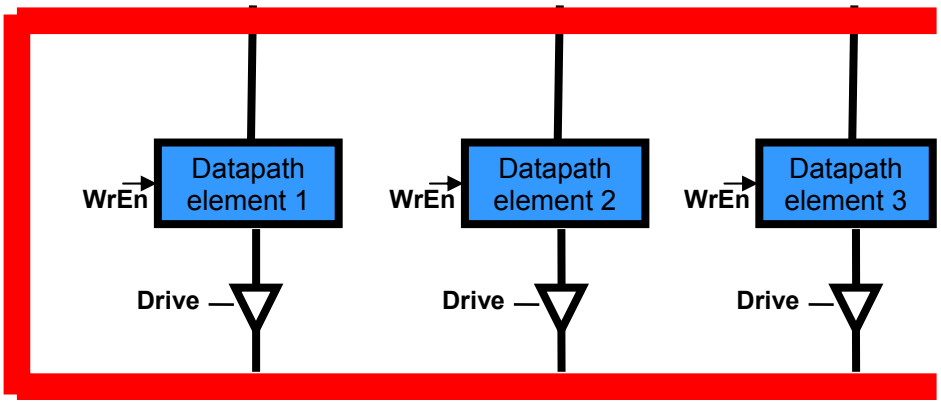


Figure 3.9 Single bus design

Bus denotes that the set of wires is shared. The first thing to notice is that the red line is a single bus (electrically), i.e., any value that is put on the bus becomes available on all segments of the wire. The second thing to notice is that there are the triangular elements that sit between the output of a datapath element and the bus. These are called *drivers* (also referred to as *tristate*[2] buffers). There is one such driver gate for each wire coming out from a datapath element that needs to be connected to a bus, and they isolate electrically the datapath element from the bus. Therefore, to electrically "connect"

[2] A binary signal is in one of two states: 0 or 1. The output of a driver when not enabled is in a third state, which is neither a 0 nor a 1. This is a high impedance state where the driver electrically isolates the bus from the datapath element that it is connected to. Hence the term tristate buffer.

datapath element 1 to the bus the associated driver must be "on". This is accomplished by selecting the "Drive" signal associated with this driver. When this is done, we say datapath element 1 is "driving" the bus. It will be a mistake to have more than one datapath element "driving" the bus at a time. So the designer of the control logic has to ensure that only one of the drivers connected to the bus is "on" in a given clock cycle. If multiple drivers are "on" at the same time, apart from the fact that the value on the bus becomes unpredictable, there is potential for seriously damaging the circuitry. On the other hand, multiple data elements may choose to grab what is on the bus in any clock cycle. To accomplish this, the WrEn (write enable) signal associated with the respective data elements (shown in Figure 3.9) has to be turned "on."

Two-bus design: Figure 3.10 illustrates a two-bus design. In this design, the register file is a dual ported one similar to that shown in Example 1. That is, two registers can be read and supplied to the ALU in the same clock cycle. Both the red (top) and purple (bottom) buses may carry address or data values depending on the need in a particular cycle. However, nominally, the red bus carries address values and the purple bus carries data values between the datapath elements.

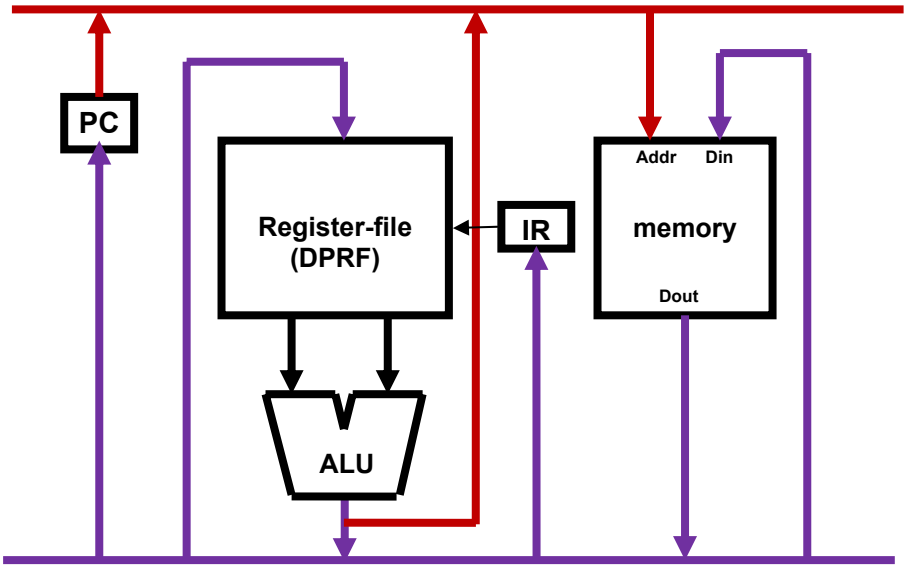


Figure 3.10 Two bus design

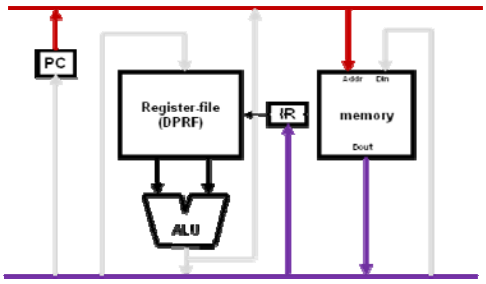
Although not shown in the Figure, it should be clear that there is a driver at the output of each of the datapath elements that are connected to either of the buses. How many cycles will be needed to carry out the Steps 1-3 mentioned mentioned at the beginning of Section 3.3.4? What needs to happen in each cycle?

Let us explore these two questions.

First clock cycle:

- PC to Red bus (note: no one else can drive the Red bus in this clock cycle)
- Red bus to Addr of Memory
- Memory reads the location specified by Addr

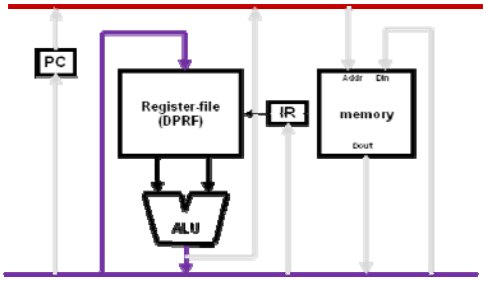
- Data from Dout to Purple bus (note: no one else can drive the Purple bus in this clock cycle)
- Purple bus to IR
- Clock IR



We have accomplished all that is needed in Steps 1 and 2 in one clock cycle.

Second clock cycle:

- IR supplies register numbers to Register file (see the dedicated wires represented by the arrow from IR to Register file); two source registers; one destination register
- Read the Register file and pull out the data from the two source registers
- Register file supplies the data values from the two source registers to the ALU (see the dedicated wires represented by the arrows from the Register file to the ALU)



- Perform the ALU ADD operation
- ALU result to Purple bus (note: no one else can drive the Purple bus in this clock cycle)
- Purple bus to Register file
- Write to the register file at the destination register number specified by IR

We have accomplished all that is needed for Step 3 in one clock cycle.

The key thing to take away is that we have accomplished steps 1-3 without having to run dedicated wires connecting every pair of datapath elements (except for the register-file to ALU connections, and the register selection wires from IR) by using the two shared buses.

3.3.6 Finite State Machine (FSM)

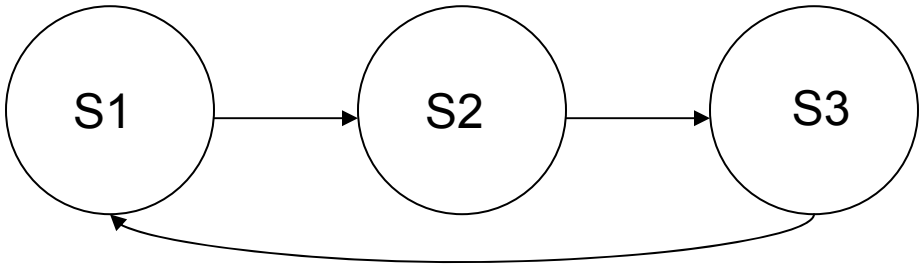


Figure 3.11: A Finite State Machine (FSM)

Thus far, we have summarized the circuit elements and how they could be assembled into a datapath for a processor. That is just one part of the processor design. The other

equally important aspect of processor design is the control unit. It is best to understand the control unit as a finite state machine, since it takes the datapath through successive stages to accomplish instruction execution.
A finite state machine, as the name suggests, has a *finite* number of states.

In Figure 3.11, S1, S2, and S3 are the *states* of the FSM. The arrows are the *transitions* between the states. FSM is an *abstraction* for any sequential logic circuit. It captures the desired behavior of the logic circuit. The state of the FSM corresponds to some physical state of the sequential logic circuit. Two things characterize a transition: (1) the external *input* that triggers that state change in the actual circuit, and (2) the *output* control signals generated in the actual circuit during the state transition. Thus, the FSM is a convenient way of capturing all the hardware details in the actual circuit.

For example, the simple FSM shown in Figure 3.12 represents the sequential logic circuit for the garage door opener that we introduced earlier. Table 3.1 shows the state transition table for this FSM with inputs that cause the transitions and the corresponding outputs they produce.

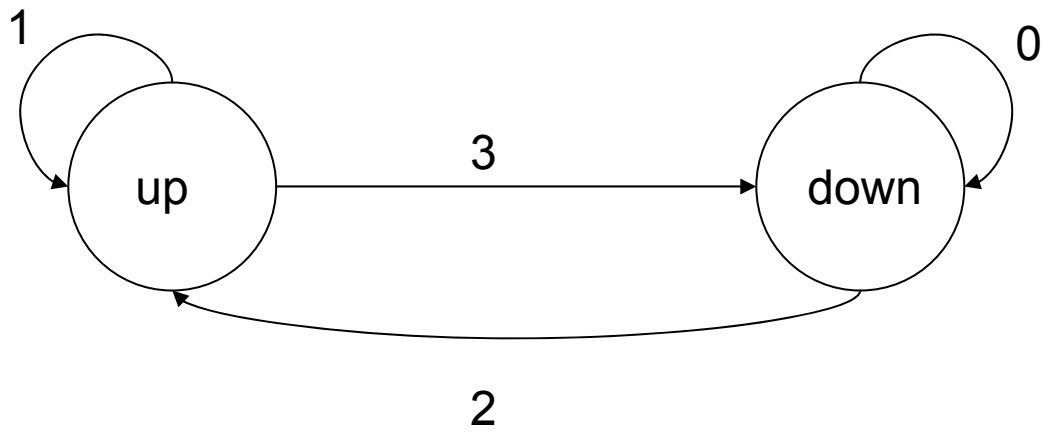


Figure 3.12: An FSM for garage door opener

Transition number	Input	State		Output
		Current	Next	
0	None	Down	Down	None
1	None	Up	Up	None
2	Clicker	Down	Up	Up motor
3	Clicker	Up	Down	Down motor

Table 3.1: State Transition Table for the FSM in Figure 3.12-(a)

The "up" state corresponds to the door being up, while the "down" state corresponds to the door being "down". The input is the clicker push-button. The outputs are the control signals to the up and down motors, respectively. The transition labeled "0" and "1" correspond to there being no clicker action. The transition labeled "2" and "3" correspond to the clicker being pushed. The former state transition ("2") is accompanied

with an "output" control signal to the up motor, while the latter ("3") is accompanied with an "output" control signal to the down motor. Anyone who has had a logic design course knows that it is a straightforward exercise to design the sequential logic circuit given the FSM and the state transition diagram (see the two exercise problems at the end of this chapter that relate to the garage door opener).

We know that sequential logic circuits can be either *synchronous* or *asynchronous*. In the former, a state transition occurs synchronized with a clock edge, while in the latter a transition occurs as soon as the input is applied.

The control unit of a processor is a sequential logic circuit as well. Therefore, we can represent the control unit by an FSM shown in Figure 3.13.

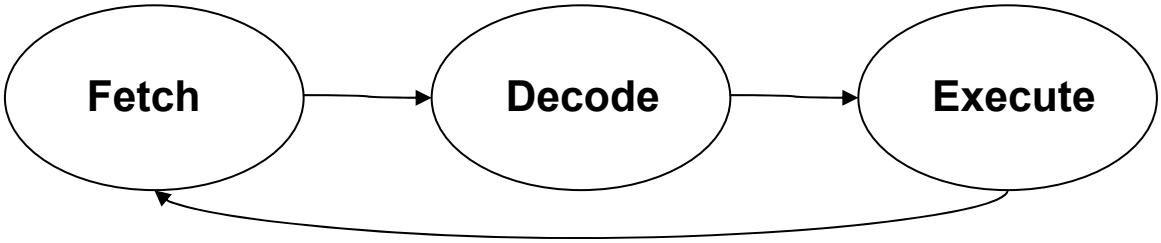


Figure 3.13: An FSM for controlling the CPU datapath

Fetch: This state corresponds to fetching the instruction from the memory.
Decode: This state corresponds to decoding the instruction brought from the memory to figure out the operands needed and the operation to be performed.
Execute: This state corresponds to carrying out the instruction execution.

We will return to the discussion of the control unit design shortly in Section 3.5.

3.4 Datapath Design

The Central Processing Unit (CPU) consists of the Datapath and the Control Unit. The datapath has all the logic elements and the control unit supplies the control signals to orchestrate the datapath commensurate with the instruction-set of the processor.

Datapath is the combination of the hardware resources and their connections. Let us review how to decide on the hardware resources needed in the datapath. As we already mentioned, the instruction-set architecture itself explicitly decides some of the hardware resources. In general, we would need more hardware resources than what is apparent from the instruction-set, as we will shortly see.

To make this discussion concrete, let us start by specifying the hardware resources needed by the instruction-set of LC-2200.

1. **ALU** capable of ADD, NAND, SUB,
2. **register file** with 16 registers (32-bit) shown in Figure 3.14
3. **PC** (32-bit)

4. **Memory** with 2^{32} X 32 bit words

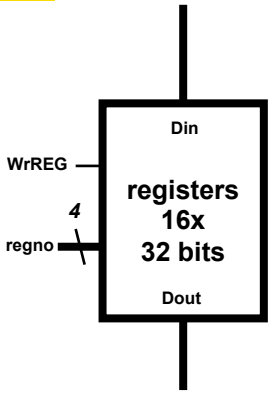


Figure 3.14: Register file with a single output port

Memory is a hardware resource that we need in LC-2200 for storing instructions and data. The size of memory is an implementation choice. The architecture only specifies the maximum size of memory that can be accommodated based on addressability. Given 32-bit addressing in LC-2200, the maximum amount of memory that can be addressed is 2^{32} words of 32-bits.

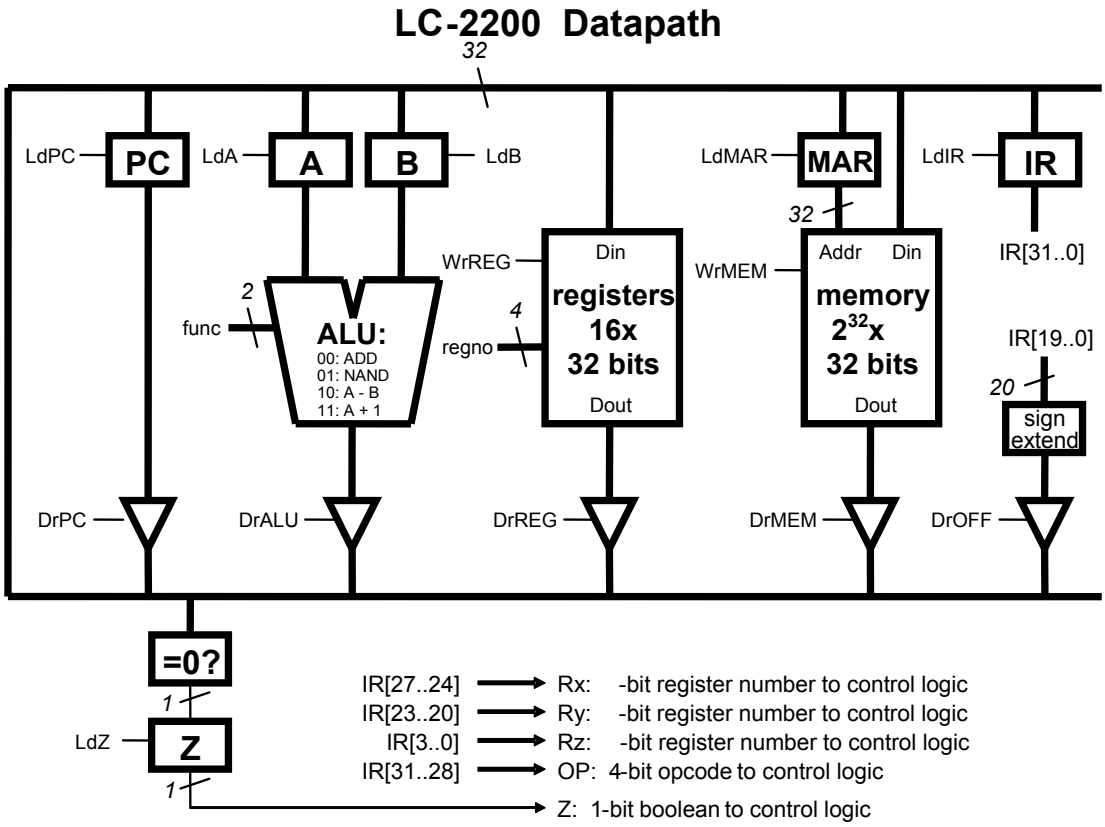


Figure 3.15: LC-2200 Datapath

Let us figure out what additional hardware resources may be needed. We already mentioned that when an instruction is brought from the memory it has to be kept in some

place in the datapath. IR serves this purpose. Let us assume we want a single bus to connect all these datapath elements. Regardless of the number of buses, one thing should be very evident right away looking at the register file. We can only get one register value out of the register file since there is only one output port (Dout). ALU operations require two operands. Therefore, we need some temporary register in the datapath to hold one of the registers. Further, with a single bus, there is exactly one channel of communication between any pair of datapath elements. This is the reason why we have **A** and **B** registers in front of the ALU. By similar reasoning, we need a place to hold the address sent by the ALU to the memory. **Memory Address Register** (**MAR**) serves this purpose. The purpose of the **Z** register (a 1-bit register) will become evident later on when we discuss the implementation of the instruction set. The zero-detect combination logic in front of the Z register (Figure 3.15) checks if the value on the bus is equal to zero. Based on the resources needed by the instruction-set, the limitations of the datapath, and implementation requirements of the instruction-set, we end up with a single bus design as shown in Figure 3.15.

3.4.1 ISA and datapath width

We have defined LC-2200 to be a 32-bit instruction-set architecture. It basically means that all instructions, addresses, and data operands are 32-bits in width. We will now explore the implication of this architectural choice on the datapath design. Let us understand the implication on the size of the buses and other components such as the ALU.

Purely from the point of view of logic design, it is conceivable to implement higher precision arithmetic and logic operations with lower precision hardware. For example, you could implement 32-bit addition using a 1-bit adder if you so choose. It will be slow but you can do it.

By the same token, you could have the bus in Figure 3.15 to be smaller than 32 bits. Such a design choice would have implications on instruction execution. For example, if the bus is just 8-bits wide, then you may have to make 4 trips to the memory to bring all 32-bits of an instruction or a memory operand. Once again, we are paying a performance penalty for making this choice.

It is a cost-performance argument that we would want to use lower precision hardware or narrower buses than what the ISA requires. The lesser the width of the buses the cheaper it is to implement the processor since most of the real estate on a chip is taken up by interconnects. The same is true for using lower precision hardware, since it will reduce the width of the wiring inside the datapath as well.

Thus, the datapath design represents a price/performance tradeoff. This is why, as we mentioned in Section 3.1, a chipmaker may bring out several versions of the same processor each representing a different point in the price/performance spectrum.

For the purposes of our discussion, we will assume that the architecture visible portions of the datapath (PC, register file, IR, and memory) are all 32-bits wide.

3.4.2 Width of the Clock Pulse

We informally discussed how to calculate the clock cycle width earlier in Section 3.3 (see Example 3). Let us formally define some of the terms involved in computing the clock cycle time.

- Every combinational logic element (for example the ALU or the drive gates in Figure 3.15) has latency for propagating a value from its input to the output, namely, *propagation delay*.
- Similarly, there is latency (called *access* time) from the time a register is enabled for reading (for example by applying a particular *regno* value to the register file in Figure 3.15) until the contents of that register appears on the output port (Dout).
- To write a value into a register, the input to the register has to be *stable* (meaning the value does not change) for some amount of time (called *set up* time) before the rising edge of the clock.
- Similarly, the input to the register has to continue to be stable for some amount of time (called *hold time*) after the rising edge of the clock.
- Finally, there is a *transmission delay* (also referred to as *wire delay*) for a value placed at the output of a logic element to traverse on the wire and appear at the input of another logic element (for example from the output of a drive gate to the input of PC in Figure 3.15).

Thus if in a single clock cycle we wish to perform a datapath action that reads a value from the register file and puts it into the A register, we have to add up all the constituent delays. We compute the worst-case delay for any of the datapath actions that needs to happen in a single clock cycle. This gives us a *lower bound* for the clock cycle time.

3.4.3 Checkpoint

So far, we have reviewed the following hardware concepts:

- Basics of logic design including combinational and sequential logic circuits
- Hardware resources for a datapath such as register file, ALU, and memory
- Edge-triggered logic and arriving at the width of a clock cycle
- Datapath interconnection and buses
- Finite State Machines

We have used these concepts to arrive at a datapath for LC-2200 instruction-set architecture.

3.5 Control Unit Design

Take a look at the picture in Figure 3.16. The role of the conductor of the orchestra is to pick out the members of the orchestra who should be playing/singing at any point of time. Each member knows what he/she has to play, so the conductor's job is essentially

keeping the timing and order, and not the content itself. If the datapath is the orchestra, then the control unit is the conductor. The control unit gives cues for the various datapath elements to carry out their respective functions. For example, if the DrALU line is asserted (i.e., if a 1 is placed on this line) then the corresponding set of driver gates will place whatever is at the output of the ALU on the corresponding bus lines.

Figure 3.16: An Orchestral arrangement[3]

Inspecting the datapath, we can list the control signals needed from the control unit:

- **Drive signals:** DrPC, DrALU, DrREG, DrMEM, DrOFF
- **Load signals:** LdPC, LdA, LdB, LdMAR, LdIR
- **Write Memory signal:** WrMEM
- **Write Registers signal:** WrREG
- **ALU function selector:** func
- **Register selector:** regno

There are several possible alternate designs to generate the control signals, all of which are the hardware realization of the FSM abstraction for the control unit of the processor.

3.5.1 ROM plus state register

[3] Source: http://www.sanyo.org.za/img/IMG_8290_nys.jpg

Let us first look at a very simple design. First, we need a way of knowing what **state** the processor is in. Earlier we introduced an FSM for the control unit consisting of the states FETCH, DECODE, and EXECUTE. These are the **macro** states of the processor in the FSM abstraction. In a real implementation, several **microstates** may be necessary to carry out the details of each of the macro states depending on the capabilities of the datapath. For example, let us assume that it takes three microstates to implement the FETCH macro states. We can then encode these microstates as

ifetch1	**0000**
ifetch2	**0001**
ifetch3	**0010**

We introduce a **state register** the contents of which hold the encoding of these microstates. So, at any instant, the contents of this register shows the state of the processor.

The introduction of the state register brings us a step closer to hardware implementation from the FSM abstraction. Next, to control the datapath elements in each microstate, we have to generate the control signals (listed above). Let us discuss how to accomplish the generation of control signals.

One simple way is to use the state register as an index into a table. Each entry of the table contains the control signals needed in that state. Returning to the analogy of an orchestra, the conductor has the music score in front of her. She maintains the "state" of the orchestra with respect to the piece they are playing. Each line of the music score tells her who all should be playing at any point of time, just as an entry in the table of the control unit signifies the datapath elements that should participate in the datapath actions in that state. The individual player knows what he/she has to play. In the same manner, each datapath element knows what it has to do. In both cases, they need someone to tell them "when" to do their act. So the job of the conductor and the control unit is exactly similar in that they give the timing necessary (the "when" question) for the players and the datapath elements to do their part at the right time, respectively. This appears quite straightforward. So let us represent each control signal by **one bit** in this table entry. If the value of the bit is one then the control signal is generated; if it is 0 then it is not generated. Of course, the number of bits in **func** and **regno** fields corresponds to their width in the datapath (2 and 4, respectively, see Figure 3.15). Figure 3.17 shows a layout of the table entry for the control signals:

	Drive Signals					Load Signals					Write Signals			
Current State	PC	ALU	Reg	MEM	OFF	PC	A	B	MAR	IR	MEM	REG	func	regno

Figure 3.17: An entry of the table of control signals

The control unit has to transition from one state to another. For example, if the FETCH macro state needs 3 micro states, then

Current state	Next state
ifetch1	ifetch2
ifetch2	ifetch3

It is relatively straightforward to accomplish this next state transition by making the next state part of the table entry. So now, our table looks as shown in Figure 3.18.

	Drive Signals					Load Signals					Write Signals				
Current State	PC	ALU	Reg	MEM	OFF	PC	A	B	MAR	IR	MEM	REG	Func	regno	Next State
...															

Figure 3.18: Next State field added to the control signals table

Let us investigate how to implement this table in hardware.

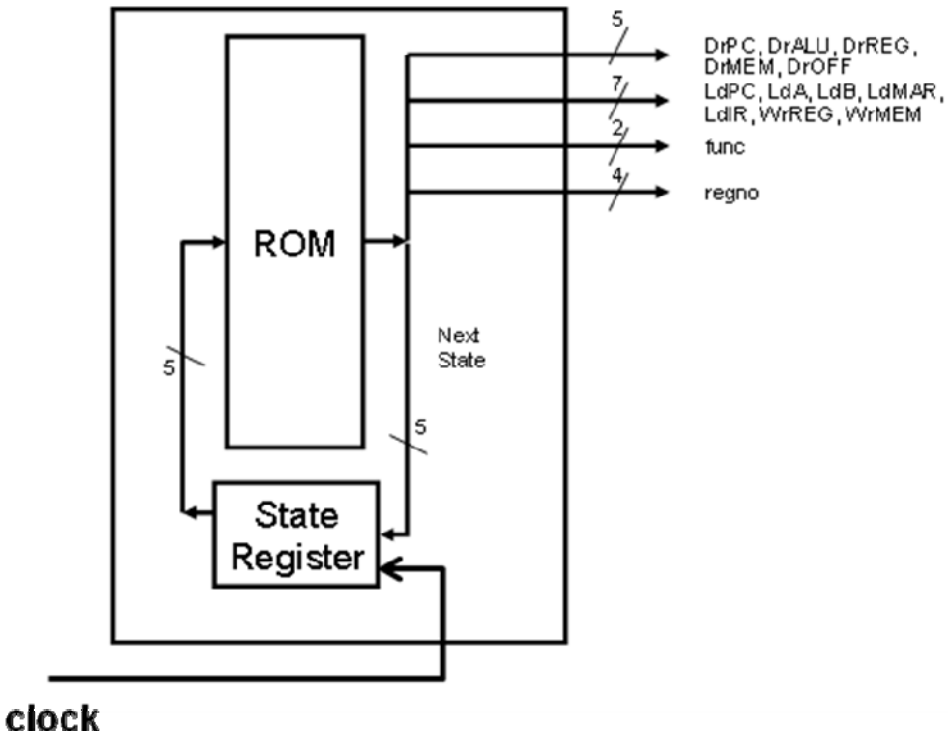


Figure 3.19: Control Unit

The table is nothing but a memory element. The property of this memory element is that once we have determined the control signals for a particular state then we can *freeze* the

contents of that table entry. We refer to this kind of memory as *Read-Only-Memory* or *ROM*.

Therefore, our hardware for the control unit looks as shown in Figure 3.19. On every clock tick, the state register advances to the next state as specified by the output of the ROM entry accessed in the current clock cycle. This is the same clock that drives all the edge-triggered storage elements in the datapath (see Figure 3.15). All the load signals coming out of the ROM (LdPC, LdMAR, etc.) serve as *masks* for the clock signal in that they determine if the associated storage element they control should be clocked in a given clock cycle.

The next thing to do is to combine the datapath and control. This is accomplished by simply *connecting* the *correspondingly named entities* in the datapath (see Figure 3.15) to the ones coming out of the ROM.

The way this control unit works is as follows:

1. The state register names the state that the processor is in for this clock cycle.
2. The ROM is accessed to retrieve the contents at the address pointed to by the state register.
3. The output of the ROM is the current set of control signals to be passed on to the datapath.
4. The datapath carries out the functions as dictated by these control signals in this clock cycle.
5. The *next state* field of the ROM feeds the state register so that it can transition to the next state at the beginning of the next clock cycle.

The above five steps are repeated in every clock cycle.

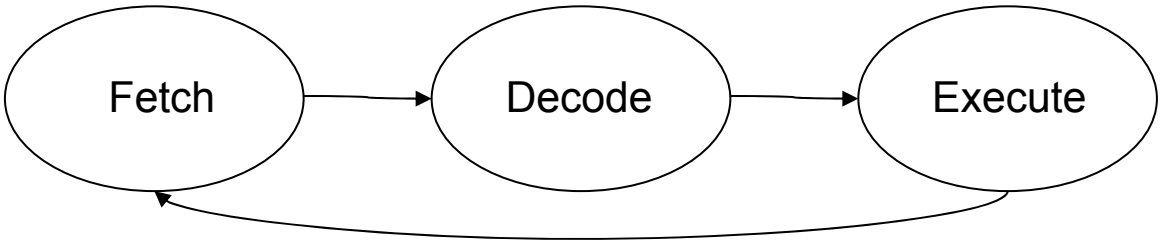


Figure 3.20: FSM for the CPU datapath reproduced

Figure 3.13 introduced the control unit of the processor as an FSM. Figure 3.20 is a reproduction of the same figure for convenience. Now let us examine what needs to happen in every macro state represented by the above FSM (Figure 3.20) and how our control unit can implement this. For each microstate, we will show the datapath actions alongside.

3.5.2 FETCH macro state

The FETCH macro state fetches an instruction from memory at the address pointed to by the Program Counter (PC) into the Instruction Register (IR); it subsequently increments the PC in readiness for fetching the next instruction.

Now let us list what needs to be done to implement the FETCH macro state:

- We need to send PC to the memory
- Read the memory contents
- Bring the memory contents read into the IR
- Increment the PC

With respect to the datapath, it is clear that with a single-bus datapath all of these steps cannot be accomplished in one clock cycle. We can break this up into the following microstates (each microstate being executed in one clock cycle):

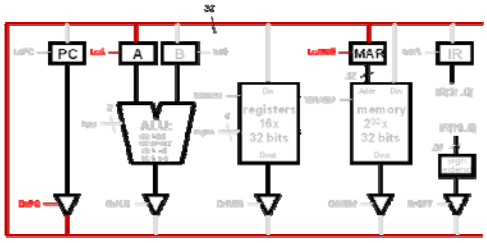
- **ifetch1**
  PC → MAR
- **ifetch2**
  MEM[MAR] → IR
- **ifetch3**
  PC → A
- **ifetch4**
  A+1 → PC

With a little bit of reflection, we will be able to accomplish the actions of the FETCH macro state in fewer than 4 cycles. Observe what is being done in **ifetch1** and **ifetch3**. The content of PC is transferred to MAR and A registers, respectively. These two states can be collapsed into a single state since the contents of PC once put on the bus can be clocked into both the registers in the same cycle. Therefore, we can simplify the above sequence to:

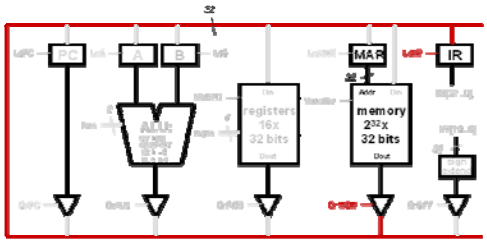
- **ifetch1**
  PC → MAR
  PC → A
- **ifetch2**
  MEM[MAR] → IR
- **ifetch3**
  A+1 → PC

Now that we have identified what needs to be done in the datapath for the microstates that implement the FETCH macro state, we can now write down the control signals needed to effect the desired actions in each of these microstates. For each microstate, we highlight the datapath elements and control lines that are activate.

- **ifetch1**
  PC → MAR
  PC → A
  Control signals needed:
  DrPC
  LdMAR
  LdA

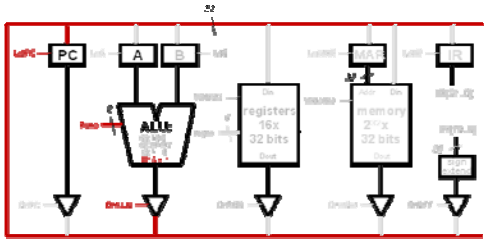


- **ifetch2**
  MEM[MAR] → IR
  Control signals needed:
  DrMEM
  LdIR



Note: with reference to the datapath, the default action of the memory is to read (i.e. when WrMEM is 0). Also the memory implicitly reads the contents of the memory at address contained in MAR and has the result available at *Dout* in **ifetch2**.

- **ifetch3**
  A+1 → PC
  Control signals needed:
  func = 11
  DrALU
  LdPC



Note: If the func selected is 11 then the ALU implicitly does A+1 (see Figure 3.15).

Now we can fill out the contents of the ROM for the addresses associated with the microstates **ifetch1**, **ifetch2**, and **ifetch3**. (X denotes "don't care"). The next-state field of **ifetch3** is intentionally marked TBD (To Be Determined), and we will come back to that shortly.

		Drive Signals					Load Signals					Write Signals				
Current State	State Num	PC	ALU	Reg	MEM	OFF	PC	A	B	MAR	IR	MEM	REG	Func	regno	Next State
Ifetch1	00000	1	0	0	0	0	0	1	0	1	0	0	0	xx	xxxx	00001
Ifetch2	00001	0	0	0	1	0	0	0	0	0	1	0	0	xx	xxxx	00010
Ifetch3	00010	0	1	0	0	0	1	0	0	0	0	0	0	11	xxxx	TBD

Figure 3.21: ROM with some entries filled in with control signals

This is starting to look like a *program* albeit at a much lower level than what we may have learned in a first computer-programming course. Every ROM location contains a set of commands that actuate different parts of the datapath. We will call each table entry a *microinstruction* and we will call the entire contents of the ROM a *micro program*. Each microinstruction also contains the (address of the) next microinstruction to be executed. Control unit design has now become a programming exercise. It is the

ultimate concurrent program since we are exploiting all the hardware concurrency that is available in the datapath in every microinstruction.

You can notice that there is a structure to each microinstruction. For example, all the drive signals can be grouped together; similarly, all the load signals can be grouped together. There are opportunities to reduce the space requirement of this table. For example, it may be possible to combine some of the control signals into one encoded field. Since we know that only one entity can drive the bus at any one time, we could group all the drive signals into one 3-bit field and each unique code of this 3-bit field signifies the entity selected to put its value on the bus. While this would reduce the size of the table, it adds a decoding step, which adds to the delay in the datapath for generating the drive control signals. We cannot group all the load signals into one encoded field since multiple storage elements may need to be clocked in the same clock cycle.

3.5.3 DECODE macro state

Once we are done with fetching the instruction, we are ready to decode it. So from the **ifetch3** microstate we want to transition to the DECODE macro state.

In this macro state, we want to examine the contents of IR (bits 31-28) to figure out what the instruction is. Once we know the instruction, then we can go to that part of the micro program that implements that particular instruction. So we can think of the DECODE process as a multi-way branch based on the OPCODE of the instruction. So we will redraw the control unit FSM depicting the DECODE as a multi-way branch. Each leg of the multi-way branch takes the FSM to the macro state corresponding to the execution sequence for a particular instruction. To keep the diagram simple, we show the multi-way branch taking the FSM to a particular class of instruction.

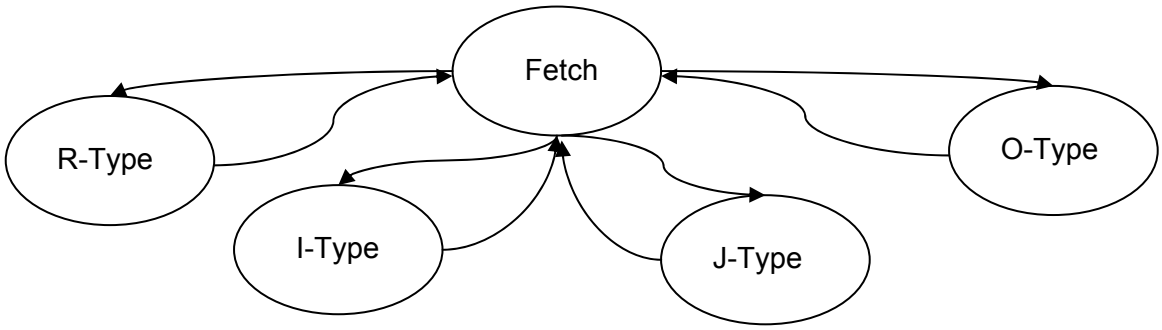


Figure 3.22: Extending the FSM with DECODE macro state fleshed out

We will come back to the question of implementing the multi-way branch in the control unit shortly. Let us first address the simpler issue of implementing each of the instructions.

3.5.4 EXECUTE macro state: ADD instruction (part of R-Type)

R-type has the following format:

Recall that the ADD instruction does the following:

$R_X \leftarrow R_Y + R_Z$

To implement this instruction, we have to read two registers from the register file and write to a third register. The registers to be read are specified as part of the instruction and are available in the datapath as the contents of IR. However, as can be seen from the datapath there is no path from the IR to the register file. There is a good reason for this omission. Depending on whether we want to read one of the source registers or write to a destination register, we need to send different parts of the IR to the **regno** input of the register-file. As we have seen **multiplexer** is the logic element that will let us do such selection.

Therefore, we add the following element (shown in Figure 3.23) to the datapath.

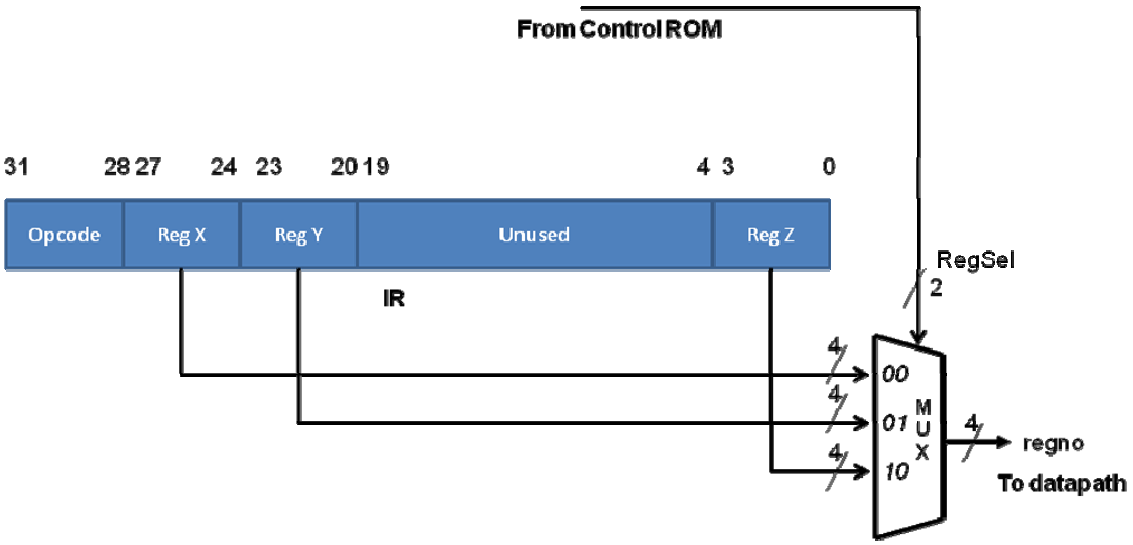


Figure 3.23: Using the IR bit fields to specify register selection

The **RegSel** control input (2-bits) comes from the microinstruction. The inputs to the multiplexer are the three different register-specifier fields of the IR (see Chapter 2 for the format of the LC-2200 instructions). It turns out that the register file is never addressed directly from the microinstruction. Therefore, we replace the 4-bit **regno** field of the microinstruction with a 2-bit **RegSel** field.

	Drive Signals					Load Signals					Write Signals				
Current State	PC	ALU	Reg	MEM	OFF	PC	A	B	MAR	IR	MEM	REG	Func	RegSel	Next State
...															

Figure 3.24: RegSel field added to the ROM control signals

Now we can write the datapath actions and the corresponding control signals needed in the microstates to implement the ADD execution macro state:

- **add1**

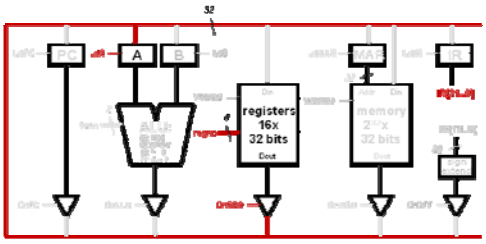
  Ry → A

  Control signals needed:

  RegSel = 01

  DrREG

  LdA



Note: The default action of the register-file is to read the contents of the register-file at the address specified by **regno** and make the data available at *Dout*.

- **add2**

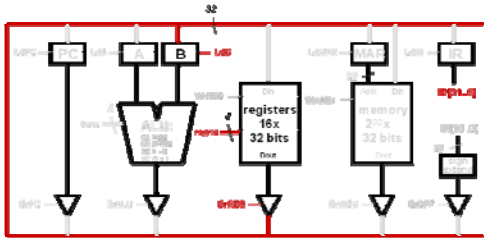
  Rz → B

  Control signals needed:

  RegSel = 10

  DrREG

  LdB



- **add3**

  A+B → Rx

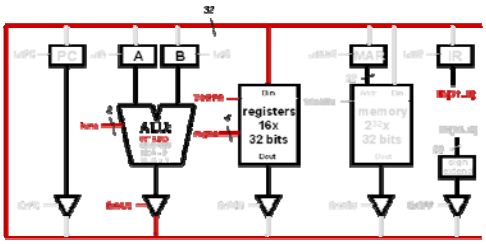
  Control signals needed:

  func = 00

  DrALU

  RegSel = 00

  WrREG



The ADD macro state is implemented by the control unit sequencing through the microstates **add1**, **add2**, **add3** and then returning to the FETCH macro state:

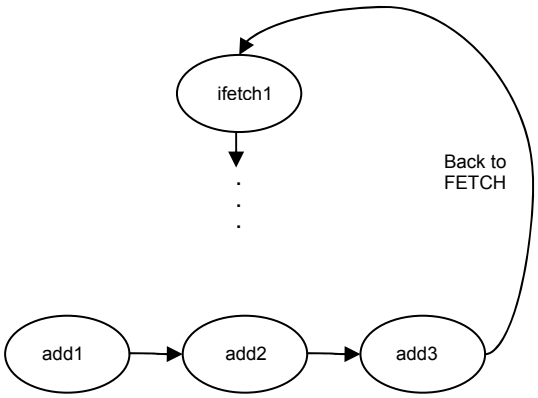


Figure 3.25: ADD macro state fleshed out in the FSM

3.5.5 EXECUTE macro state: NAND instruction (part of R-Type)

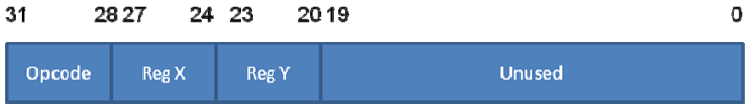
Recall that the NAND instruction does the following:

$R_X \leftarrow R_Y$ NAND R_Z

The NAND macro state is similar to ADD and consists of **nand1**, **nand2**, and **nand3** microstates. We leave it as an exercise to the reader to figure out what changes are needed in those microstates compared to the corresponding states for ADD.

3.5.6 EXECUTE macro state: JALR instruction (part of J-Type)

J-type instruction has the following format:



Recall that JALR instruction was introduced in LC-2200 for supporting the subroutine calling mechanism in high-level languages. JALR, stashes the return address in a register, and transfers control to the subroutine by doing the following:

$R_Y \leftarrow PC + 1$

$PC \leftarrow R_X$

Given below are the microstates, datapath actions, and control signals for the JALR instruction:

- **jalr1**

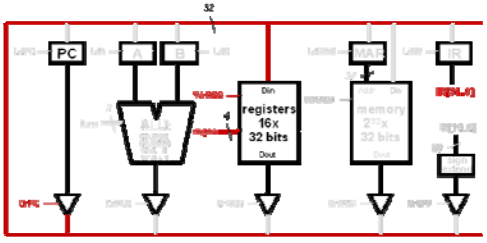
  PC → Ry

  Control signals needed:

  DrPC

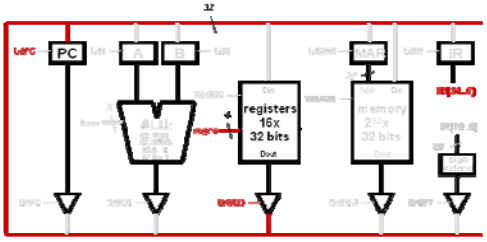
  RegSel = 01

  WrREG



Note: PC+1 needs to be stored in Ry. Recall that we already incremented PC in the FETCH macro state.

- **jalr2**
  Rx → PC
  Control signals needed:
  RegSel = 00
  DrREG
  LdPC



3.5.7 EXECUTE macro state: LW instruction (part of I-Type)

I-type instruction has the following format:

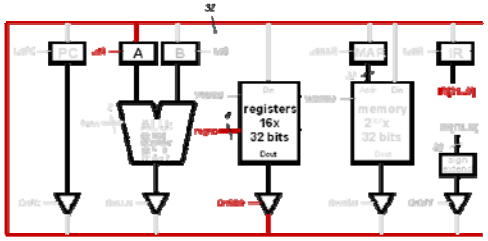
Recall that LW instruction has the following semantics:

$R_X \leftarrow$ MEMORY[R_Y + signed address-offset]

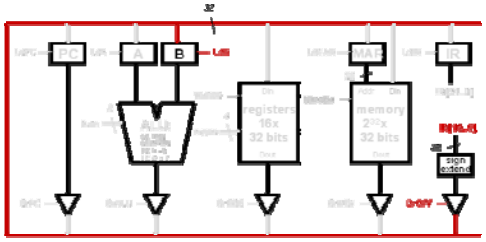
In the I-Type instruction, the signed address offset if given by an immediate field that is part of the instruction. The immediate field occupies IR 19-0. As can be seen from the datapath, there is a **sign-extend** hardware that converts this 20-bit 2's complement value to a 32-bit 2's complement value. The **DrOFF** control line enables this sign-extended offset from the IR to be placed on the bus.

Given below are the microstates, datapath actions, and control signals for LW instruction:

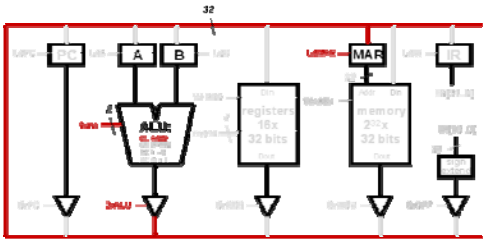
- **lw1**
  Ry → A
  Control signals needed:
  RegSel = 01
  DrREG
  LdA



- **lw2**
  Sign-extended offset → B
  Control signals needed:
  DrOFF
  LdB



- **lw3**
  A+B → MAR
  Control signals needed:
  func = 00
  DrALU
  LdMAR



- **lw4**

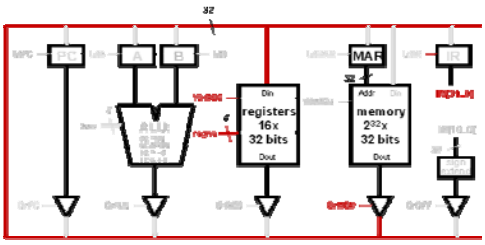
  MEM[MAR] → Rx

  Control signals needed:

  DrMEM

  RegSel = 00

  WrREG

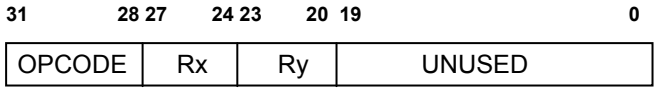


Example 4:

We have decided to add another addressing mode **autoincrement** to LC-2200. This mode comes in handy for LW/SW instructions. The semantics of this addressing mode with LW instruction is as follows:

```
LW    Rx, (Ry)+     ;     Rx <- MEM[Ry];
                    ;     Ry <- Ry + 1;
```

The instruction format is as shown below:



Write the sequence for implementing the LW instruction with this addressing mode (you need to write the sequence for the execution macro state of the instruction). For each microstate, show the datapath action (in register transfer format such as A←Ry) along with the control signals you need to enable for the datapath action (such as DrPC).

Answer:

LW1: Ry -> A, MAR
Control Signals:
RegSel=01; DrReg; LdA; LdMAR

LW2: MEM[Ry] -> Rx
Control Signals:
DrMEM; RegSel=00; WrREG

LW3: A + 1 -> Ry
Control Signals:
Func=11; DrALU; RegSel=01; WrREG

3.5.8 EXECUTE macro state: SW and ADDI instructions (part of I-Type)

Implementation of the SW macro state is similar to the LW macro state. Implementation of the ADDI macro state is similar to the ADD macro state with the only difference that the second operand comes from the immediate field of the IR as opposed to another

register. The development of the microstates for these two instructions is left as an exercise to the reader.

3.5.9 EXECUTE macro state: BEQ instruction (part of I-Type)

BEQ instruction has the following semantics:

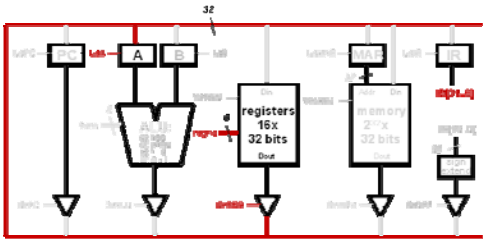
If (R_X == R_Y) then PC ← PC + 1 + signed address-offset
else nothing

This instruction needs some special handling. The semantics of this instruction calls for comparing the contents of two registers (Rx and Ry) and branching to a target address generated by adding the sign-extended offset to PC+1 (where PC is the address of the BEQ instruction) if the two values are equal.

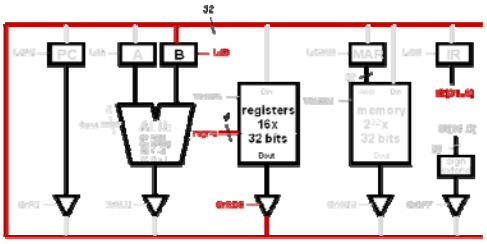
In the datapath, there is hardware to detect if the value on the bus is a zero. The microstates for BEQ use this logic to set the Z register upon comparing the two registers.

Given below are the microstates, datapath actions, and control signals for the BEQ macro state:

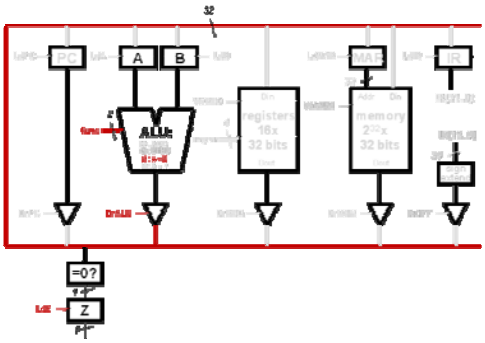
- **beq1**
  Rx → A
  Control signals needed:
  RegSel = 00
  DrREG
  LdA



- **beq2**
  Ry → B
  Control signals needed:
  RegSel = 01
  DrREG
  LdB



- **beq3**
  A – B
  Load Z register with result of zero detect logic
  Control signals needed:
  func = 10
  DrALU
  LdZ

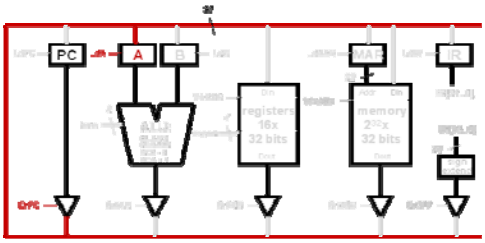


Note: The zero-detect combinational logic element in the datapath is always checking if the value on the bus is zero. By asserting LdZ, the Z register (1-bit register) is capturing the result of this detection for later use.

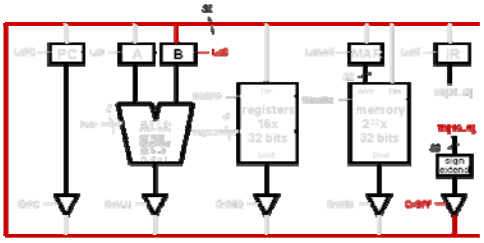
The actions following this microstate get tricky compared to the micro states for the other instructions. In the other instructions, we simply sequenced through all the microstates for that instruction and then return to the FETCH macro state. However, BEQ instruction causes a control flow change depending on the outcome of the comparison. If the Z register is not set (i.e., Rx != Ry) then we simply return to **ifetch1** to continue execution with the next sequential instruction (PC is already pointing to that instruction). On the other hand, if Z is set then we want to continue with the microstates of BEQ to compute the target address of branch.

First let us go ahead and complete the microstates for BEQ assuming a branch has to be taken.

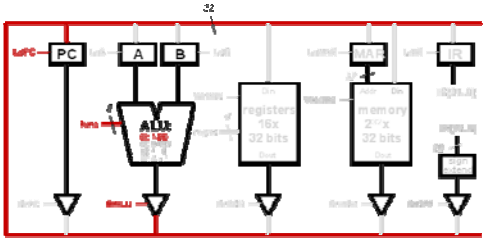
- **beq4**
  PC→ A
  Control signals needed:
  DrPC
  LdA



- **beq5**
  Sign-extended offset → B
  Control signals needed:
  DrOFF
  LdB



- **beq6**
  A+B → PC
  Control signals needed:
  func = 00
  DrALU
  LdPC



Note: In the FETCH macro state itself, PC has been incremented. Therefore, we simply add the sign-extended offset to PC to compute the target address.

3.5.10 Engineering a conditional branch in the microprogram

Figure 3.26 shows the desired state transitions in the BEQ macro state. The **next-state** field of the **beq3** microinstruction will contain **beq4**. With only one **next-state** field in the microinstruction, we need to engineer the transition from **beq3** to **ifetch1** or **beq4** depending on the state of the Z register. A time-efficient way of accomplish this task is by using an additional location in the ROM to duplicate the microinstruction corresponding to **ifetch1**. We will explain how this is done.

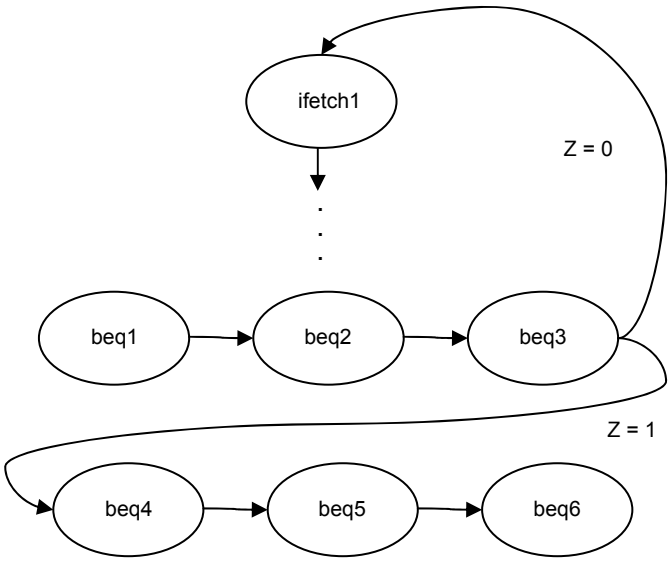


Figure 3.26: Desired state transitions in the BEQ macro state

Let us assume the state register has 5 bits, **beq4** has the binary encoding **0**1000, and the next state field of the **beq3** microinstruction is set to **beq4.** We will prefix this encoding with the contents of the Z register to create a 6-bit address to the ROM. If the Z bit is 0 then the address presented to the ROM will be **0**01000 and if the Z bit is 1 then the address will be **1**01000. The latter address (101000) is the one where we will store the microinstruction corresponding to the **beq4** microstate. In the location **0**01000 (let us call this **ifetch1-clone**) we will store the exact same microinstruction as in the original **ifetch1** location. Figure 3.27 shows this pictorially.

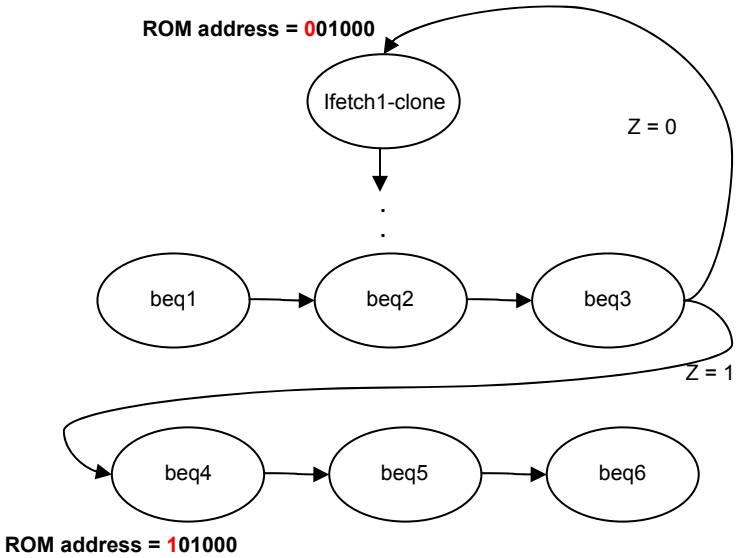


Figure 3.27: Engineering the conditional micro branch

We can extend this idea (of cloning a microinstruction) whenever we need to take a conditional branch in the micro program.

3.5.11 DECODE macro state revisited

Let us return to the DECODE macro state. Recall that this is a multi-way branch from **ifetch3** to the macro state corresponding to the specific instruction contained in IR. We adopt a trick similar to the one with the Z register for implementing the 2-way branch for the BEQ instruction.

Let us assume that 10000 is the encoding for the *generic* EXECUTE macro state.

The **next-state** field of **ifetch3** will have this generic value. We prefix this generic value with the contents of the OPCODE (IR bits 31-28) to generate a 9-bit address to the ROM. Thus, the ADD macro state will start at ROM address **0000**10000, the NAND macro state will start at ROM address **0001**10000, the ADDI at ROM address **0010**10000, and so on.

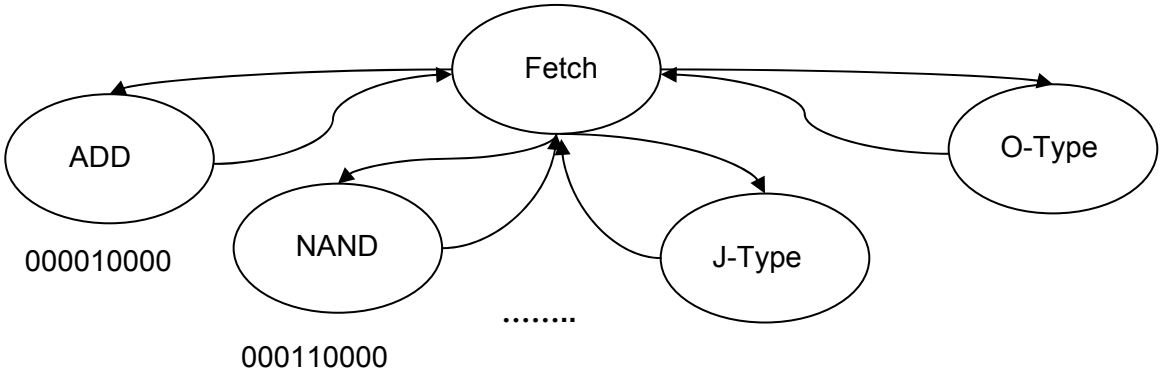


Figure 3.28: Effecting the multi-way branch of DECODE macro state

Putting all this together, the control logic of the processor has a 10-bit address (as shown in Figure 3.29):

- **top 4 bits from IR 31-28**
- **next bit is the output of the Z register**
- **the bottom 5 bits are from the 5-bit state register**

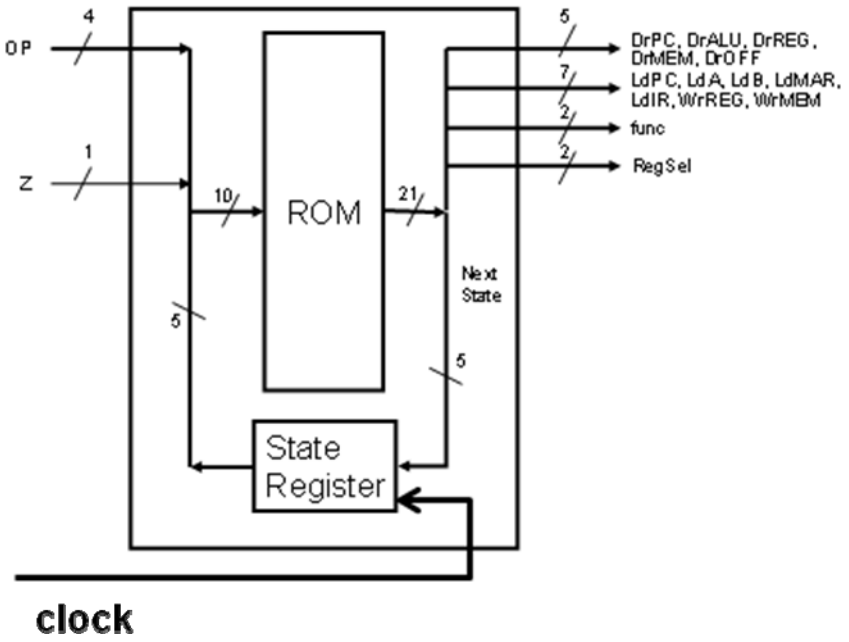


Figure 3.29: LC-2200 Control Unit

The lines coming in and out of the above control unit connect directly to the corresponding signal lines of the datapath in Figure 3.15.

3.6 Alternative Style of Control Unit Design

Let us consider different styles of implementing the control unit of a processor.

3.6.1 Microprogrammed Control

In Section 3.5, we presented the microprogrammed style of designing a control unit. This style has a certain elegance, simplicity, and ease of maintenance. The advantage we get with the micro programmed design is the fact that the control logic appears as a program all contained in a ROM and lends itself to better maintainability. There are two potential sources of inefficiency with microprogrammed design. The first relates to **time**. To generate the control signals in a particular clock cycle, an address has to be presented to the ROM and after a delay referred to as the **access time** of the ROM; the control signals are available for operating on the datapath. This time penalty is in the critical path of the clock cycle time and hence is a source of performance loss. However, the time penalty can be masked by employing prefetching of the next microinstruction while the current one is being executed. The second source of inefficiency relates to **space**. The design presented in the previous section represents one specific style of microprogramming called *horizontal microcode*, wherein there is a bit position in each microinstruction for every control signal needed in the entire datapath. From the sequences we developed so

far, it is clear that in most microinstructions most of the bits are 0 and only a few bits are 1 corresponding to the control signals that are needed for that clock cycle. For example, in **ifetch1** only **DrPC**, **LdMAR**, and **LdA** are 1. All the other bits in the microinstruction are 0. The space inefficiency of horizontal microcode could be overcome by a technique called *vertical microcode* that is akin to writing assembly language programming. Basically, the bit position in each microinstruction may represent different control signals depending on an opcode field in each vertical microinstruction. Vertical microcode is trickier since the control signals corresponding to a particular bit position in the microinstruction have to be mutually exclusive.

3.6.2 Hardwired control

It is instructive to look at what exactly the ROM represents in the horizontal microcode developed in Section 3.5. It really is a **truth table** for all the control signals needed in the datapath. The rows of the ROM represent the states and the columns represent the functions (one for each control signal). From previous exposure to logic design, the reader may know how to synthesize the minimal Boolean logic function for each column. Such logic functions are more efficient than a truth table.

We can implement the Boolean logic functions corresponding to the control signals using combinational logic circuits. The terms of this function are the conditions under which that control signal needs to be asserted.

For example, the Boolean function for **DrPC** will look as follows:

DrPC = ifetch1 + jalr1 + beq4 + …

Using AND/OR gates or universal gates such as NAND/NOR all the control signals can be generated. We refer to this style of design as **hardwired control** since the control signals are implemented using combinational logic circuits (and hence hardwired, i.e, they are not easy to change). Such a design leads to both time (no access time penalty for ROM lookup) and space (no wasted space for signals that are NOT generated in a clock cycle) efficiency. There has been some criticism in the past that such a design leads to a maintenance nightmare since the Boolean functions are implemented using random logic.

However, the advent of **Programmable Logic Arrays (PLAs)** and **Field Programmable Gate Arrays (FPGAs)** largely nullifies this criticism. For example, PLAs give a structure to the random logic by organizing the required logic as a structured 2-dimensional array. We show this organization in Figure 3.30.

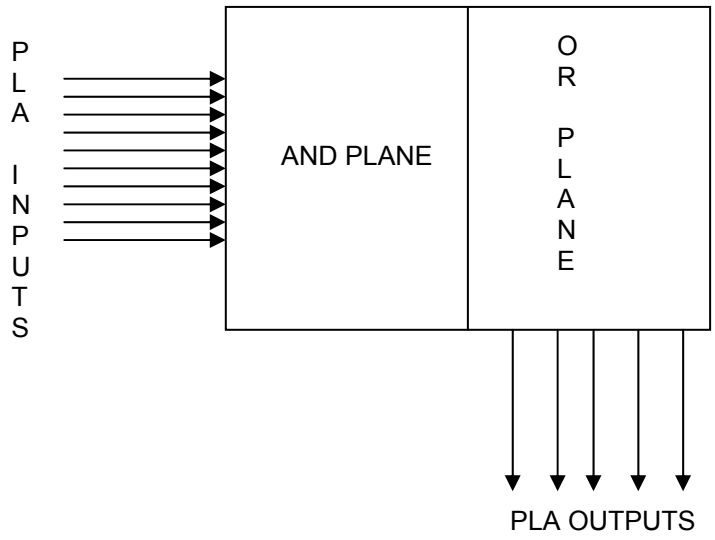


Figure 3.30: PLA

The outputs are the control signals needed to actuate the datapath (**DrPC, DrALU,** etc.). The inputs are the state the processor is in (**ifetch1, ifetch2,** etc.) and the conditions generated in the datapath (contents of the IR, Z, etc.). Every gate in the AND plane has ALL the inputs (true and complement versions). Similarly every gate in the OR plane has as inputs ALL the product terms generated by the AND plane. This is called a PLA because the logic can be "programmed" by selecting or not selecting the inputs to the AND and OR planes. Every PLA OUTPUT is a **SUM-of-PRODUCT** term. The PRODUCT terms are implemented in the AND plane. The SUM terms are implemented in the OR plane. This design concentrates all the control logic in one place and thus has the structural advantage of the micro programmed design. At the same time since the control signals are generated using combinational logic circuits, there is no penalty to this hardwired design. There is a slight space disadvantage compared to a true random logic design since every AND gate has a fan-in equal to ALL the PLA INPUTS and every OR gate has a fan-in equal to the number of AND gates in the AND plane. However, the structural advantage and the regularity of the PLA lend itself to compact VLSI design and far out-weigh any slight disadvantage.

More recently, FPGAs have become very popular as a way of quickly prototyping complex hardware designs. An FPGA is really a successor to the PLA and contains logic elements and storage elements. The connections among these elements can be programmed "in the field" and hence the name. This flexibility allows any design bugs to be corrected more easily even after deployment thus improving the maintainability of hardwired design.

3.6.3 Choosing between the two control design styles

The choice for control design between the two styles depends on a variety of factors.. We have given the pros and cons of both control styles. It would appear with the advent of FPGAs, much of the maintainability argument against hardwired control has

disappeared. Nevertheless, for basic implementation of processors (i.e., non-pipelined) as well as for the implementation of complex instructions (such as those found in the Intel x86 architecture), microprogrammed control is preferred due to its flexibility and amenability to quick changes. On the other hand, as we will see in the chapter on pipelined processor design, hardwired control is very much the preferred option for high-end pipelined processor implementation. Table 3.2 summarizes the pros and cons of the two design approaches.

Control Regime	Pros	Cons	Comment	When to use	Examples
Microprogrammed	Simplicity, maintainability, flexibility Rapid prototyping	Potential for space and time inefficiency	Space inefficiency may be mitigated with vertical microcode Time inefficiency may be mitigated with prefetching	For complex instructions, and for quick non-pipelined prototyping of architectures	PDP 11 series, IBM 360 and 370 series, Motorola 68000, complex instructions in Intel x86 architecture
Hardwired	Amenable for pipelined implementation Potential for higher performance	Potentially harder to change the design Longer design time	Maintainability can be increased with the use of structured hardware such as PLAs and FPGAs	For High performance pipelined implementation of architectures	Most modern processors including Intel Xeon series, IBM PowerPC, MIPS

Table 3.2: Comparison of Control Regimes

3.7 Historical Perspective

It is instructive to look back in time to see how the performance tradeoffs relate to economic and technological factors.

In the 1940's & 1950's, logic elements for constructing the hardware and memory were extremely expensive. Vacuum tubes and later discrete transistors served as the implementation technology. The instruction-set architecture was very simple, and typically featured a single register called an *accumulator*. Examples of machines from this period include EDSAC and IBM 701.

In the 1960's, we started seeing the first glimmer of "integration." IBM 1130 that was introduced in 1965 featured *Solid Logic Technology (SLT)* that was a precursor to integrated circuits. This decade saw a drop in hardware prices but memory was still

implemented using magnetic cores (called *core memory*) and was a dominant cost of a computer system.

The trend continued in the 1970's with the initial introduction of SSI (Small Scale Integrated) and then MSI (Medium Scale Integrated) circuits as implementation technologies for the processor. Semiconductor memories started making their way in the 70's as a replacement for core memory. It is interesting to note that circa 1974 the price per bit for core memory and semiconductor memory were about equal ($0.01 per bit). Semiconductor memory prices started dropping rapidly from then on, and the rest is history!

The 70's was an era for experimenting with a number of different architecture approaches (such as stack-oriented, memory-oriented, and register-oriented). Examples of machines from this era include IBM 360 and DEC PDP-11 for a hybrid memory- and register-oriented architecture; and Burroughs B-5000 for a stack-oriented approach. All of these are variants of a *stored program computer*, which is often referred to as *von Neumann* architecture, named after John von Neumann, a computer pioneer.

In parallel with the development of the stored program computer, computer scientists were experimenting with radically new architectural ideas. These include *dataflow* and *systolic* architectures. These architectures were aimed at giving the programmer control over performing several instructions in parallel, breaking the mold of sequential execution of instructions that is inherent in the stored program model. Both dataflow and systolic architectures focus on data rather than instructions. The dataflow approach allows all instructions whose input data is ready and available to execute and pass the results of their respective executions to other instructions that are waiting for these results. The systolic approach allows parallel streams of data to flow through an array of functional units pre-arranged to carry out a specific computation on the data streams (e.g., matrix multiplication). While the dataflow architecture is a realization of a general model of computation, systolic is an algorithm-specific model of synthesizing architectures. While these alternative styles of architectures did not replace the stored program computer, they had tremendous impact on computing as a whole all the way from algorithm design to processor implementation.

In the 1980's, there were several interesting developments. In the first place, high-speed LSI (Large Scale Integrated) circuits using bipolar transistors were becoming commonplace. This was the implementation technology of choice for high-end processors such as IBM 370 and DEC VAX 780. In parallel with this trend, VLSI (Very Large Scale Integrated) circuits using CMOS transistors (also called Field Effect Transistors or FETs) were making their way as vehicles for single-chip microprocessors. By the end of the decade, and in the early 1990's these *killer micros* started posing a real threat to high-end machines in terms of price/performance. This decade also saw rapid advances in compiler technologies and the development of a true partnership between system software (as exemplified by compilers) and instruction-set design. This partnership paved the way for RISC (Reduced Instruction Set Computer) architectures. IBM 801, Berkeley RISC, and Stanford MIPS processors led the way in the RISC

revolution. Interestingly, there was still a strong following for the CISC (Complex Instruction Set Computer) architecture as exemplified by Motorola 68000 and Intel x86 series of processors. The principle of pipelined processor design (see Chapter 5) which until now was reserved for high-end processors made its way into microprocessors as the level of integration allowed placing more and more transistors in one piece of silicon. Interestingly, the debate over RISC versus CISC petered out and the issue became one of striving to achieve an instruction throughput of one per clock cycle in a pipelined processor.

The decade of the 1990 has firmly established the Microchip (single chip microprocessors based on CMOS technology) as the implementation technology of choice in the computer industry. By the end of the decade, Intel x86 and Power PC instruction-sets (curiously, both are CISC style architectures that incorporated a number of the implementation techniques from the RISC style architectures) became the industry standard for making "boxes," be they desktops, servers, or supercomputers. It is important to note that one of the most promising architectures of this decade was DEC Alpha. Unfortunately, due to the demise of DEC in the late 90's, the Alpha architecture also rests in peace!

With the advent of personal communication devices (cell phones, pagers, and PDAs) and gaming devices, embedded computing platforms have been growing in importance from the 80's. Interestingly, a significant number of embedded platforms use descendents of a RISC architecture called ARM (Acorn RISC Machine, originally designed by Acorn Computers). Intel makes XScale processors that are derived from the original ARM architecture. A point to note is that ARM processors were originally designed with PCs and workstations in mind.

Superscalar and *VLIW* (Very Large Instruction Word architectures) processors represent technologies that grew out of the RISC revolution. Both are attempts to increase the throughput of the processor. They are usually referred to as *multiple issue* processors. Superscalar processor relies on the hardware to execute a fixed number of mutually independent adjacent instructions in parallel. In the VLIW approach, as the name suggests, an instruction actually contains a number of operations that are strung together in a single instruction. VLIW relies heavily on compiler technology to reduce the hardware design complexity and exploit parallelism. First introduced in the late 80's, the VLIW technology has gone through significant refinement over the years. The most recent foray of VLIW architecture is the IA-64 offering from Intel targeting the supercomputing marketplace. VLIW architectures are also popular in the high-end embedded computing space as exemplified by DSP (Digital Signal Processing) applications.

We are reaching the end of the first decade of the new millennium. There is little debate over instruction-sets these days. Most of the action is at the level of micro-architecture, namely, how to improve the performance of the processor by various hardware techniques. Simultaneously, the level of integration has increased to allow placing

multiple processors on a single piece of silicon. Dubbed *multicore*, such chips have started appearing now in most computer systems that we purchase today.

3.8 Review Questions

1. What is the difference between level triggered logic and edge triggered logic? Which do we use? Why?

2. Given the FSM and state transition diagram for a garage door opener (Figure 3.12-a and Table 3.1) implement the sequential logic circuit for the garage door opener (Hint: The sequential logic circuit has 2 states and produces three outputs, namely, next state, up motor control and down motor control).

3. Re-implement the above logic circuit using the ROM plus state register approach detailed in this chapter.

4. Compare and contrast the various approaches to control logic design.

5. One of the optimizations to reduce the space requirement of the control ROM based design is to club together independent control signals and represent them using an encoded field in the control ROM. What are the pros and cons of this approach? What control signals can be clubbed together and what cannot be? Justify your answer.

6. What are the advantages and disadvantages of a bus-based datapath design?

7. Consider a three-bus design. How would you use it for organizing the above datapath elements? How does this help compared to the two-bus design?

8. To save time would it be possible to store intermediate values in registers on the datapath such as the Program Counter or Instruction Register? Explain why or why not.

9. The Instruction Fetch is implemented in the text with first 4 states and then three. What would have to be done to the datapath to make it two states long?

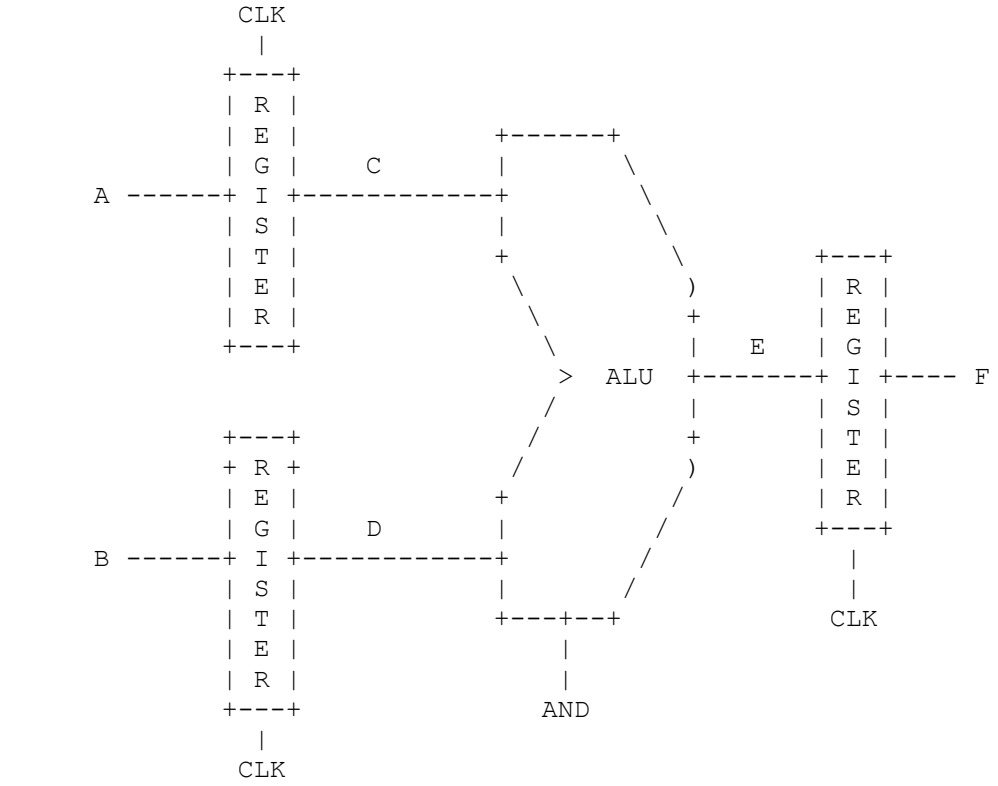
10. How many words of memory will this code snippet require when assembled? Is space allocated for "L1"?

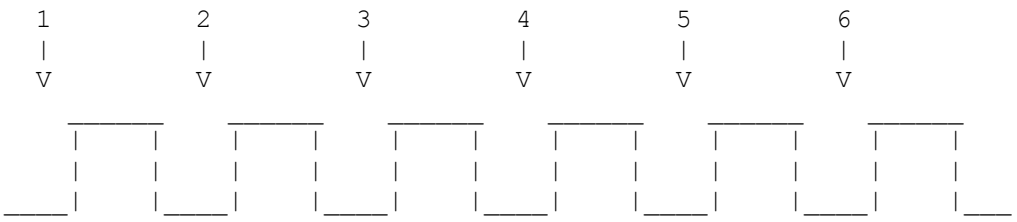
```
        beq $s3, $s4, L1
        add $s0, $s1, $s2
L1:     sub $s0, $s0, $s3
```

11. What is the advantage of fixed length instructions?

12. What is a leaf procedure?

13. For this portion of a datapath (assuming that all lines are 16 bits wide) Fill in the table below.





Time	A	B	C	D	E	F
1	0x42	0xFE	0	0	0	0
2	0	0				
3	0xCAFE	0x1				
4	0	0				
5	0	0				
6	0	0				

14. Suppose you are writing an LC-2200 program and you want to jump a distance that is farther that allowed by a BEQ instruction. Is there a way to jump to an address?

15. Could the LC series processors be made to run faster if it had a second bus? If your answer was no, what else would you need?

    Another ALU
    An additional mux
    A TPRF

16. Convert this statement:

    g = h + A[i];

    into an LC-2200 assembler with the assumption that the Address of A is located in $t0, g is in $s1, h is in $s2, and, i is in $t1

17. Suppose you design a computer called the Big Looper 2000 that will never be used to call procedures and that will automatically jump back to the beginning of memory when it reaches the end. Do you need a program counter? Justify your answer.

18. In the LC-2200 processor, why is there not a register after the ALU?

19. In the datapath diagram shown in Figure 3.15, why do we need the A and B registers in front of the ALU? Why do we need MAR? Under what conditions would you be able to do without with any of these registers? [Hint: Think of additional ports in the register file and/or buses.]

20. Core memory used to cost $0.01 per bit. Consider your own computer. What would be a rough estimate of the cost if memory cost is $0.01/bit? If memory were still at that price what would be the effect on the computer industry?

21. If computer designers focused entirely on speed and ignored cost implications, what would the computer industry look like today? Who would the customers be? Now consider the same question reversed: If the only consideration was cost what would the industry be like?

22. Consider a CPU with a stack-based instruction set. Operands and results for arithmetic instructions are stored on the stack; the architecture contains no general-purpose registers.

    The data path shown on the next page uses two separate memories, a 65,536 (216) byte memory to hold instructions and (non-stack) data, and a 256 byte memory to hold the stack. The stack is implemented with a conventional memory and a stack pointer register. The stack starts at address 0, and grows upward (to higher addresses) as data are pushed onto the stack. The stack pointer points to the element

on top of the stack (or is -1 if the stack is empty). You may ignore issues such as stack overflow and underflow.

Memory addresses referring to locations in program/data memory are 16 bits. All data are 8 bits. Assume the program/data memory is byte addressable, i.e., each address refers to an 8-bit byte. Each instruction includes an 8-bit opcode. Many instructions also include a 16-bit address field. The instruction set is shown below. Below, "memory" refers to the program/data memory (as opposed to the stack memory).

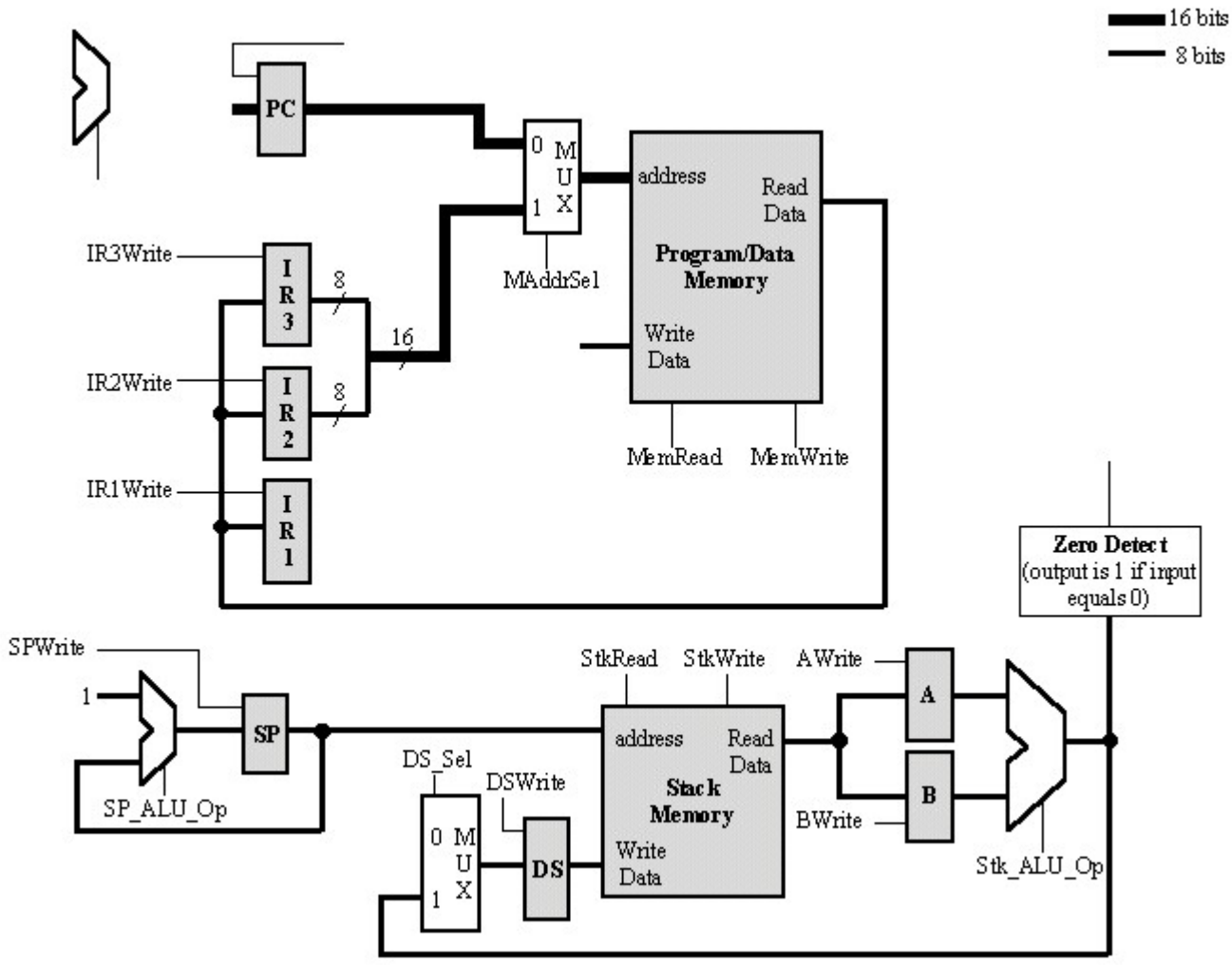
OPCODE	INSTRUCTION	OPERATION
00000000	PUSH <addr>	push the contents of memory at address <addr> onto the stack
00000001	POP <addr>	pop the element on top of the stack into memory at location <addr>
00000010	ADD	Pop the top two elements from the stack, add them, and push the result onto the stack
00000100	BEQ <addr>	Pop top two elements from stack; if they're equal, branch to memory location <addr>

Note that the ADD instruction is only 8 bits, but the others are 24 bits. Instructions are packed into successive byte locations of memory (i.e., do NOT assume all instruction uses 24 bits).

Assume memory is 8 bits wide, i.e., each read or write operation to main memory accesses 8 bits of instruction or data. This means the instruction fetch for multi-byte instructions requires multiple memory accesses.

Datapath

Complete the partial design shown on the next page.



Assume reading or writing the program/data memory or the stack memory requires a single clock cycle to complete (actually, slightly less to allow time to read/write registers). Similarly, assume each ALU requires slightly less than one clock cycle to complete an arithmetic operation, and the zero detection circuit requires negligible time.

Control Unit

Show a state diagram for the control unit indicating the control signals that must be asserted in each state of the state diagram.