

DIGITAL SYSTEM DESIGN BASICS**1. Introduction**

A High-Level Language statement ($A = B + C$) specifies an **operation** (an addition). A compiler translates the statement to a few machine language instructions. A machine language instruction specifies **architectural operations**. Thus, a high-level language operation is implemented by a number of architectural operations.

The architecture is realized by the organization, based on the speed, cost, power, etc. goals. The organization has resources that are grouped into modules called **digital systems**. For example, a computer consists of *at least* three digital systems : a CPU, a memory and an I/O controller. A digital system performs **microoperations** and so contains registers, buses, Arithmetic Logic Units (ALUs) and a sequencer. Note that every computer today has a CPU and the microprocessor chip contains the CPU. For example, if one buys a PC with a Pentium IV microprocessor, the microprocessor chip has the Intel Pentium IV CPU (in addition to cache memories, buffers, etc.).

The CPU, being a digital system, consists of registers, buses, ALUs and a sequencer ! It runs (executes) machine language instructions by performing **microoperations**. That is, the CPU performs architectural operations of a machine language instruction by performing a number of **microoperations**. Therefore, an architectural operation is implemented by a few microoperations on the microarchitecture layer. Then, the conclusion is that **a machine language program is run by performing microoperations** ! The CPU accesses the memory that keeps instructions and data. Thus, to run instructions, the CPU and memory interact. However, the CPU is the more active module during the instruction execution. The I/O controllers are needed to move data between the memory and I/O devices, such as the disk, keyboard, mouse, printer, etc.

Architectural and organizational decisions determine which microoperations are needed. Certainly, a complex architectural operation will require more/complex microoperations. Similarly, a complex machine language instruction set will result many/complex microoperations. Thus, a CISC CPU performs more/complex microoperations than a RISC CPU. Typical microoperations of a computer are fetching an instruction, incrementing the program counter, calculating the effective address, adding two registers, reading data from the memory, writing data to the memory, etc.

The organization level is realized by the logic level. The logic level consists of **digital circuits**. A digital circuit contains gates and flip-flops. Then, a digital system is made up of digital circuits. Designing digital circuits requires digital logic knowledge, i.e. Switching Algebra, formal sequential circuit synthesis rules, etc. The logic level is realized by the transistor level where the resources are digital integrated circuits, containing transistors, resistors, capacitors and diodes. Then, a digital system is made up of transistors, resistors, capacitors and diodes. The design of the transistor level requires electrical engineering knowledge.

2. Digital System Design

Our earliest experimentation with digital systems started in the 18th century when first steam-operated mechanical textile machines were manufactured and continued with the electromechanical telephone exchange systems of the early 20th century. Today's digital systems are more complex than ever. For example, a microprocessor has hundreds of millions to billions of transistors. But, there are still no formal techniques to design digital systems today. That is, there is **no** specific algebra or formal circuit synthesis rules. Therefore, digital system design is based on the divide-and-conquer method where a digital system is repeatedly partitioned into smaller and smaller pieces until each is reduced to a small digital circuit upon which Switching Algebra and sequential circuit synthesis rules are applied.

For today's high-density chips, such as microprocessors, help is needed and are available for the complex digital system design process. Methods and tools have been developed, accomplishing two seemingly contradictory jobs : hiding details and simultaneously allowing the design to freely move from one level to another (simultaneous design of the levels) down to the transistor level. One such digital system design technique that borrows ideas from sequential

circuit synthesis rules, is the **finite-state machine (FSM)** technique. A state diagram with finite number of states is drawn to show which microoperation happens when and how, thus describing both the datapath and the control unit. In this technique, **a FSM state diagram describes a digital system.**

Another design technique, currently used by industry is **Hardware Description Language- (HDL-)** based. It is needed when a complex circuit is designed. HDL statements have constructs to describe hardware events, resulting in the abstraction of hardware. Such statements are as understandable as high-level language statements. In this technique, **a HDL program describes a digital system.** Another HDL program describes another digital system. Most commonly used HDLs are VHDL and the Verilog HDL. Designers work on these HDL programs and delay the complex task of building the prototype until they are sure about their HDL program. Polytechnic has a number of courses that use the HDL approach, for example EL 547, EL 549 and EL 644.

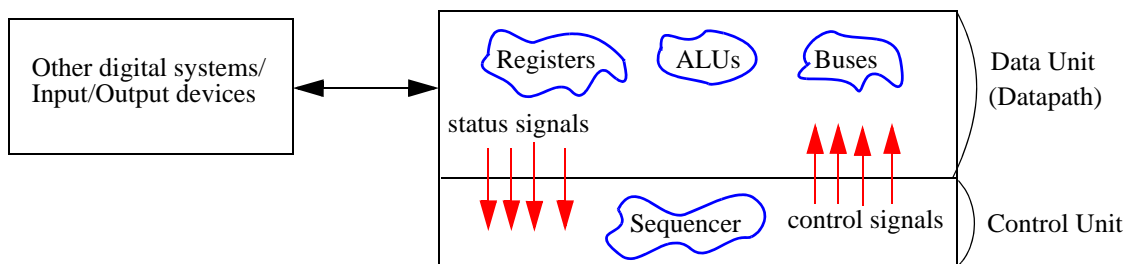
FSM and HDL approaches lend themselves to computer usage for speedy correct design. For example, verifying that the design works is done by obtaining correct results from the simulations of the FSM or HDL design on a computer. Such simulations are fast, allowing designers to start the prototype building phase quickly. For complex circuit design, HDL languages together with software tools are powerful enough to describe a digital system at the microarchitecture, logic and transistor layers, alleviating the lack of formalism in the digital system design to a certain extent.

In CS 2214, the finite state machine design technique will be used. We will concentrate mostly on the CPU and obtain a state diagram only for the CPU. A state diagram will describe the MIPS CPU digital system, specifying which microoperation happens when and how. The other digital systems (the memory and I/O controllers) will be designed partially, without obtaining their state diagrams. Below, first, a brief summary of the FSM technique and datapath construction are introduced. Then an introduction to digital circuits is given.

2.1. High-Level Digital System Design

Digital System design process includes a series of partitionings today. However, there are no specific rules about how to partition a digital system today, except that the **first** partitioning of every digital system is into **data** and **control** units. This partitioning is *universal*. The data unit, also known as **datapath**, performs the microoperations. The control unit determines the timing and sequence of the microoperations (the complex task of determining which microoperation happens when). Thus, the control unit controls the data unit. Then, one can partition in a number of different ways, each i) simplifying or complicating the design process and each, ii) decreasing or increasing the cost, speed, size, power consumption, reliability, etc. Thus, in digital system design, there are plenty of good designs, each satisfying a different set of speed/cost/size/power consumption/reliability/... goals.

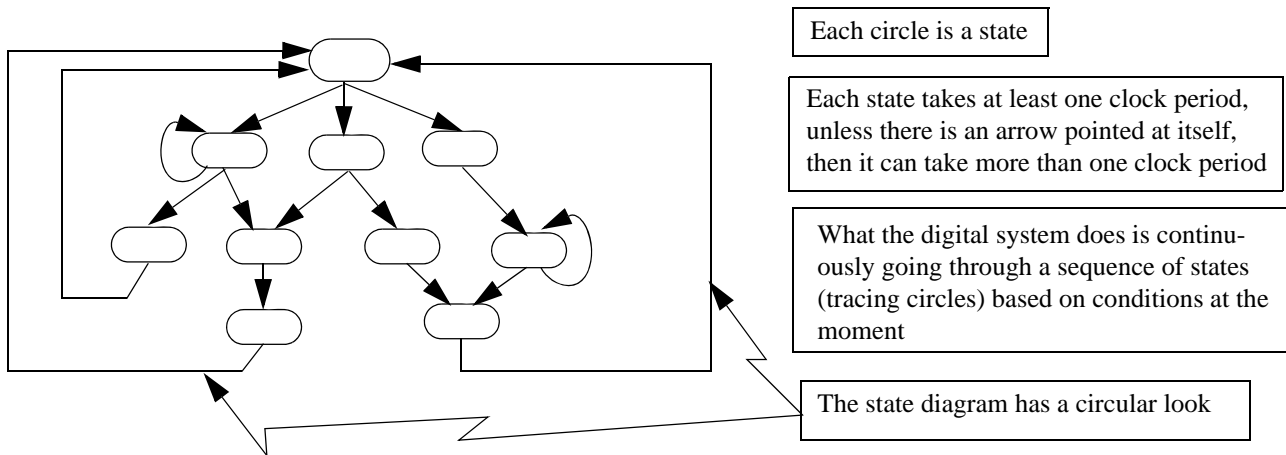
The data unit performs microoperations such as additions, subtractions, AND, OR, shift, compare, fetch an instruction, read data from memory, write data to the memory. In order to perform these microoperations, it needs to have three kinds of hardware : registers, ALUs and buses as shown below. Registers are needed to keep data temporarily, ALUs perform additions, subtractions, shifts, ANDs, ORs, etc. and buses interconnect registers and ALUs. Registers contain flip-flops to store bits. ALUs are often combinational circuits, i.e. they contain only gates. Buses are bundles of wires with additional control logic. The control unit consists of a sequencer that generates control signals based on the status signals from the datapath. It is the sequencer that determines the sequence of microoperations.



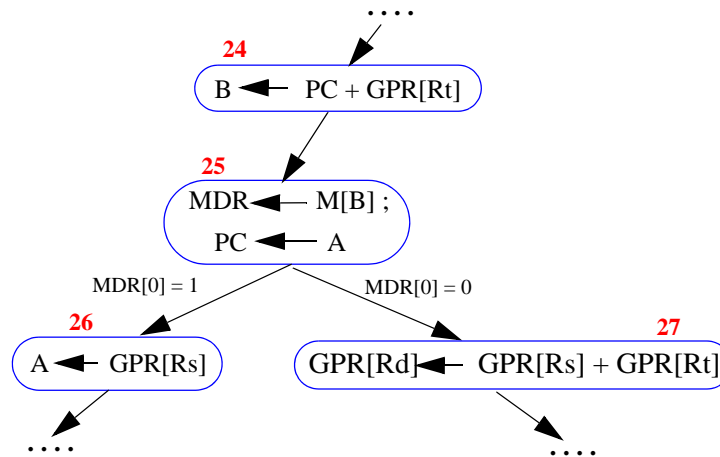
While the control unit seems to perform rather a simple job of controlling the data unit and the data unit has most of the logic, the datapath is easier to design than the control unit because of two reasons. One is that the components are not ideal. They have gate delays and fan-in restrictions and consume power. Often, the control unit circuit logically works but when it becomes operational, it does not work due to **glitches** created by the gate delays. The second reason is that the data unit is highly **regular**. That is, a data unit has pieces of hardware repeated many times. For example, an ADDer is a repetitive set of smaller addition blocks, the multiplier is similar and so are registers. Another example is that a 4-bit compare circuit can be repeated 8 times to compare 32-bit numbers. The design, test, modify, manufacture, upgrade of regular hardware is easier. It also costs less. But, the control unit is not that regular. The sequencer is implemented by either hardwiring or microprogramming or a mixture of both. Especially, hardwired control units have considerable random logic that their design is difficult.

The FSM techniques requires that we obtain a state diagram with a finite number of states. Each state specifies zero, one or more microoperations and how what they require in hardware. Thus, the FSM state diagram describes the digital system. However, the FSM technique is preferable if the number of states is small. That is, it is practical when the digital system is relatively simple. Otherwise, one would have to use the HDL technique. In CS2214, we will use the FSM technique since the CPU digital system we implement executes a small subset of the instructions discussed in class. However, the FSM diagram is such that it is easy to modify it to execute additional instructions.

In the FSM technique, the state diagram has rather a “circular” look where the same states are traced one after another as shown below. Sometimes, a few states can be different based on the conditions at the moment. One can interpret the FSM state diagram as a “**graphical program**” where each circle is a “program line” and the branches coming out of a state represent an “If statement.” Since the state diagram is circular, one can think of the digital system in an “infinite loop” performing same microoperations again and again. In the case of a CPU digital system, the state diagram is traced top to bottom **once for each instruction**. If the CPU executes 20 machine language instructions, it will trace the state diagram top to bottom 20 times.



Below, a partial picture of the state diagram of an **imaginary** digital system is shown. Again, each circle in a state diagram represents a **state** and each state corresponds to a **clock period**. In order to identify states easily, they are numbered as shown above : state 24, state 25, etc. There is **no** relationship between a state number and the clock period the state occurs. For example, state 24 above, might or might not occur in clock period 24. **Only one (1) state occurs in one clock period**. Assume that for the remaining portion of the handout, state 24 occurs in clock period 46. In clock period 47, state 25 will occur... There may be multiple branches coming out of a state, as in state 25 above. Only one of the states will happen in the next clock period. State 25 occurs in clock period 47, then, in clock period 48, either state 26 or state 27 will occur, but **not** both.



Often a microoperation moves a value from a register to another register as in state 26. This is called a **register transfer**. A typical state diagram may have many register transfers. That is why we sometimes name the microarchitecture level, the **Register Transfer Level (RTL)**. It is the same reason why the microoperation notation used in the above figure ($PC \leftarrow A$) is called the *Register Transfer Language*.

Each state has zero, one or more **microoperations**. State 24 has one microoperation, so does state 26 :

$B \leftarrow PC + GPR[Rt]$; A register add microoperation in state 24

$A \leftarrow GPR[Rs]$; A register transfer microoperation in state 26

If a state has more than one microoperation, they happen **in parallel**. Thus, the order of writing these microoperations in a state does **not** matter. For example, state 25 has two microoperations that happen in parallel :

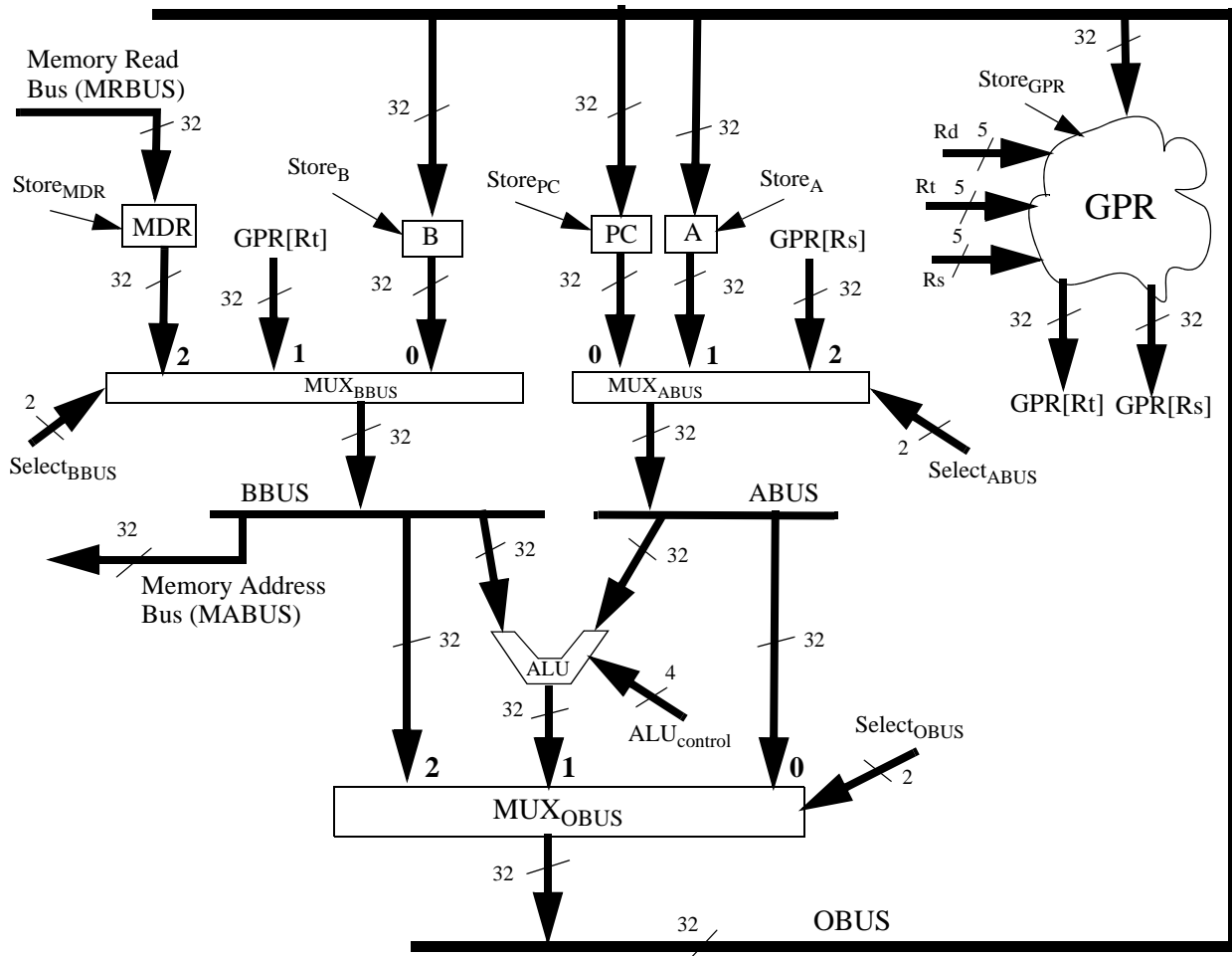
$MDR \leftarrow M[B]$; A memory read microoperation in state 25

$PC \leftarrow A$; A register transfer microoperation also in state 25

The process of determining which branch to take is **testing**. In the above picture, the **rightmost** bit of register MDR is tested to take a branch : $MDR[0]$. This test is performed in state **25**, in parallel with the memory read. The testing is done by the control unit, while the datapath transfers the value from the memory to the MDR. Thus, the control unit checks the current value of MDR, not the value read from the memory at the same moment. The testing determines which one of the microoperations will happen in the next clock period (clock period 48). In order to test MDR_0 , this leftmost bit is input from the datapath to the control unit as a **status** signal. If an instruction is tested in a clock period, we call it **instruction decoding**. Above, we might or might not be decoding an instruction.

2.2. The Datapath Design

As stated earlier, the datapath performs microoperations for which it has to have registers, buses and ALUs. The transfers from a source to a destination in the state diagram are typically carried out by **buses** in a digital system. Otherwise, there must be direct connections between every source and destination pair which is very expensive. This is the reason for buses. Typically, a datapath contains at least **three internal** buses to interconnect the registers and ALUs. The figure below has three internal buses : ABUS, BBUS and OBUS. The Memory Data Bus (MDB) and the Memory Address Bus (MABUS) connected to the memory are **external** buses.

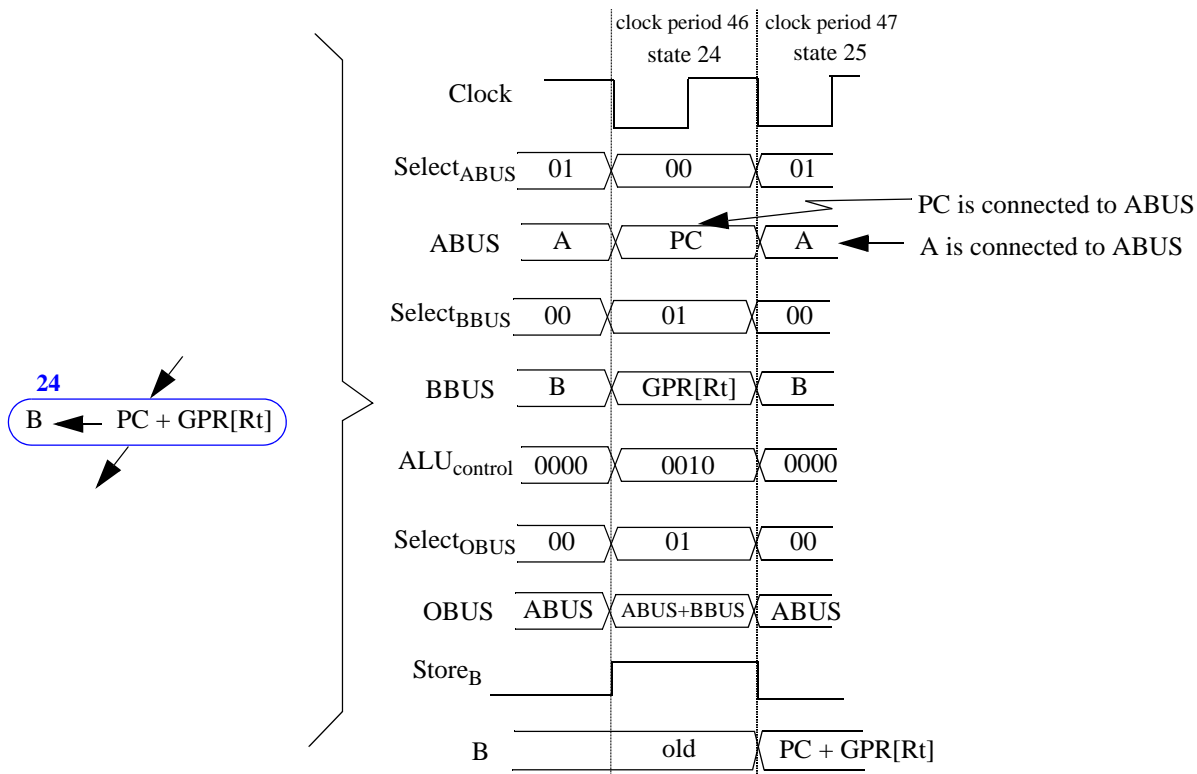


A bus is a **shared** set of lines. A bus has two or more sources and at least one destination. The ABUS above has three sources (PC, A and GPR[Rs]) and two destinations (an ALU input and OBUS). A bus is constructed by connecting the sources to a multiplexer (MUX) whose output is the bus. Select signals of the MUX connect a source to the bus. Select_{ABUS} lines connect source PC to ABUS if they are 00. They connect source A to ABUS, if they are 01. If they are 10, they connect GPR[Rs] to ABUS. The Select_{ABUS} signals are control signals and generated by the control unit. If a MUX has more than two sets of data inputs, it will have more than one select line. All the buses above have more than two sets of data lines input. For example, OBUS has eight different sets of data inputs and so has three select lines. Note that we will draw control signals as **angled** lines to make sure we know they are generated by the control unit.

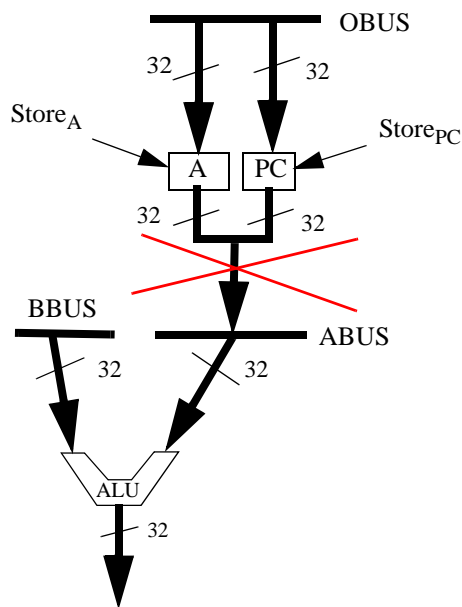
A bus gets a value as soon as it is “connected” a value, which happens in the beginning of the clock period the bus is used. For example, the ABUS is connected PC in state 24. ABUS gets the value of PC in the beginning of clock period 46 as shown below. It is important to note that **a bus does not remember** what was connected to it in the past. For example, in clock period 47, the ABUS does not remember the value of PC connected to it in clock period 46.

The reason why we use a multiplexer to form a bus is that we use **totem-pole** gates and FFs (registers are totem-pole registers) as opposed to **tri-state** or **wired-OR** gates and FFs. It is important **NOT** to short-circuit the outputs of totem-pole gates and FFs as done below. A MUX solves the problem.

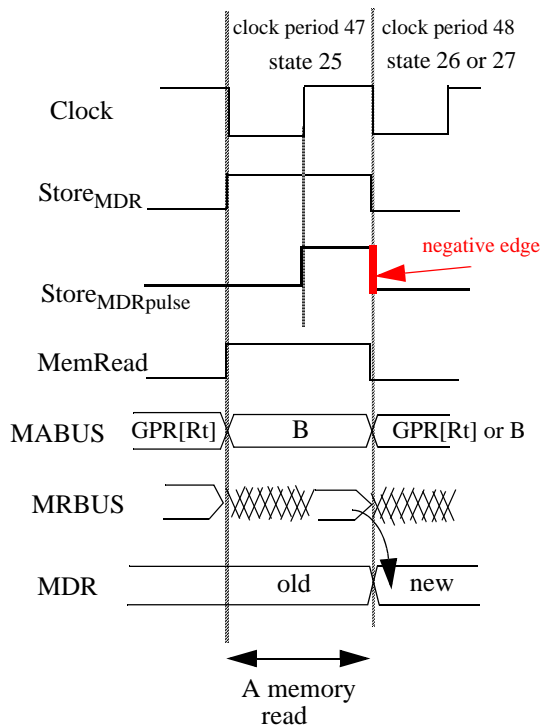
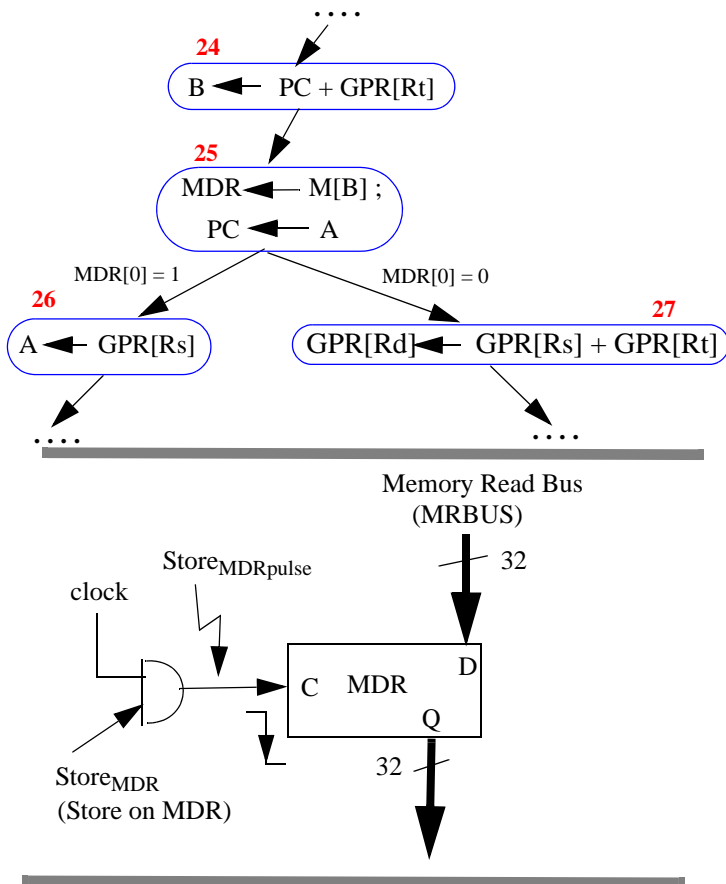
A datapath has registers to keep (remember) values for the future. Storing a value on a register is very different from connecting a value to a bus. A register which is transferred a value in clock period “x” will actually get the new value



in the beginning of the following clock period : "x + 1." For example, register B in state 24 (clock period 46) is transferred a new value which is (PC + GPR[Rt]). The B register takes on the new value in the beginning of clock period 47 (in state 25), **not** in clock period 46 (not in state 24).



Similarly, register MDR and PC take on their new values in the beginning of clock period 48 (one of states 26 or 27), not in clock period 47 (not in state 25). Thus, the MDR[0] bit that is tested in state 25 is **not** the new bit, but the old one. The reason why a register takes on (is stored) a new value in the beginning of the next clock period is that flip-flops of registers are clocked at the **end** of the clock period.



The reason why flip-flops (FFs) are clocked at the end of a particular clock period is that they are **edge-triggered** FFs. In CS2214, we frequently use negative-edge triggered FFs : they need negative edges to store values.

When we want to store a value on a register in a state, say, state 25 which occurs in clock period 47, a negative edge, $STORE_{MDRpulse}$, is supplied to the clock (C) input of the register in state 25 as shown on the left.

The $Store_{MDR}$ signal is active, 1, when we want to store on MDR, that is, in state 25. The AND of the clock signal and the $Store_{MDR}$ signal is $Store_{MDRpulse}$ that results in a **negative edge**. The negative edge happens at the end of the 47th clock period, storing the Memory Read Bus (MRBUS) value in the beginning of the 48th clock period.

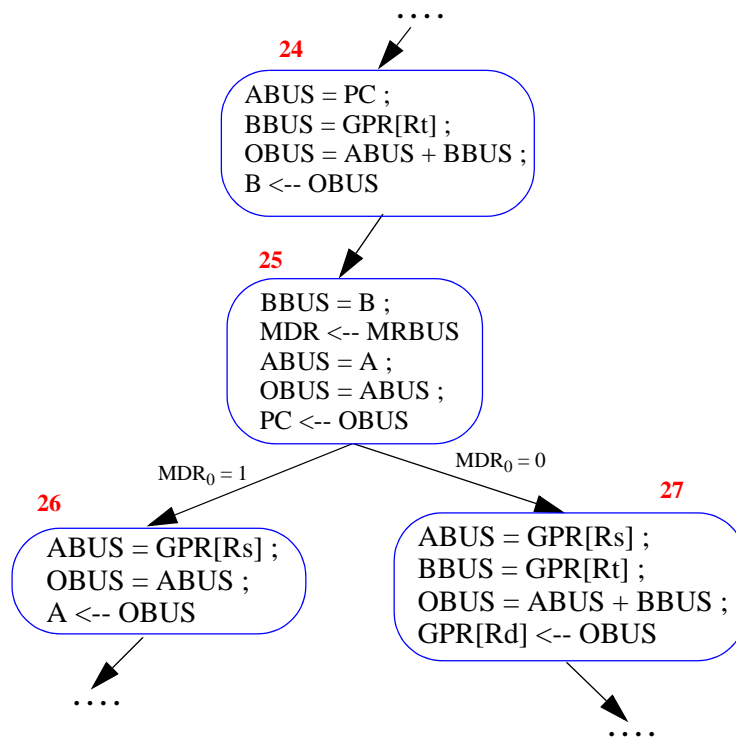
Note that the MDR register is in the datapath. The data input to MDR is from the memory : MDB, carries the content of a memory location. The $Store_{MDR}$ signal is generated by the control unit of the digital system and is supplied to the datapath.

The memory operation happens as follows : In the beginning of the 47th clock period, the CPU activates its read signal to the memory called MemRead. The memory uses the Memory Address Bus (MABUS) as the address to access the location. Before the end of the clock period the memory delivers the content of the location which is stored at the end of clock period 47. Note that the value of MRBUS in the beginning of clock period 47 changes (is not stable) as the memory is in the process of accessing the location. Eventually, the location is accessed and the value becomes stable.

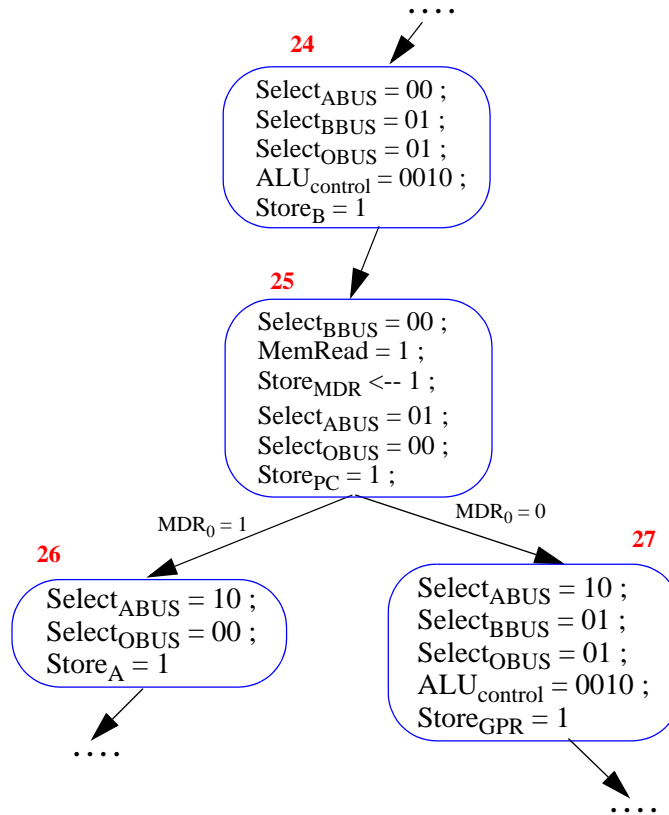
Finally, until we discuss Chapter 7, the Memory Hierarchy chapter, we will assume that we can read or write a memory location in one clock period as seen on the left. Similarly, we will assume we have hardware fast enough to add two registers in one clock period as implied by states 24 and 27 in the state diagram.

The finite state diagram above is called the high-level state diagram. In the diagram, the data movements are **not** written in terms of buses. This is common. When we start the design, we typically obtain a state diagram without buses. We just show register transfers. Then, we decide about the busing structure, such as how many buses and which register is a source on which bus. These decisions are based on the speed, cost, power, size, etc. goals. For example, the more buses a digital system has, the higher the speed and the cost. This is because, with more buses, we can have more data transfers in parallel, not in sequence, saving time. But, with more buses, there are more wires and control logic and so the cost is high. In addition, the placement of registers on buses is not straightforward. For example, in state 24 we add PC and GPR[Rt] which means PC and GPR[Rt] cannot be on the same bus to do the addition. They must be connected to different buses. In state 27, we add GPR[Rs] and GPR[Rt], meaning that they must be on separate buses to do the addition.

Let's now convert the above high-level state diagram to the one with buses. Note that we use an **arrow** if the destination is a register and an **equal** sign if the destination is a bus or wire. As stated above, all microoperations in a state happen at the same time. This means, all the connections and transfers happen simultaneously in a state, not sequentially. So, we can write the connections and transfers in a state in any order, it does not matter.

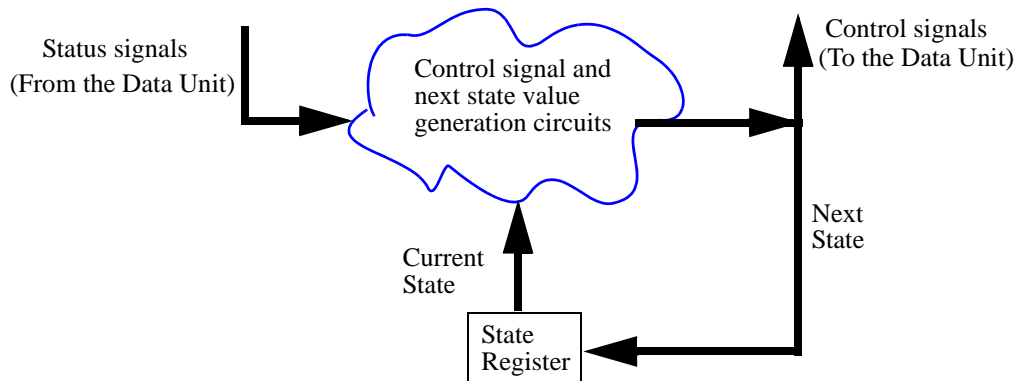


Next step is obtaining a state diagram where the microoperations are described in terms of control signals. That state diagram is the low-level state diagram. These control signals are determined after the datapath design is complete and when the control unit is designed. It is the control unit that has to generate these control signals and so its design depends on this control signal determination very much. However, the control unit has to receive status signals from the datapath as well. The status signals are also determined during the control unit design. Below, we show the low-level state diagram of the above system.

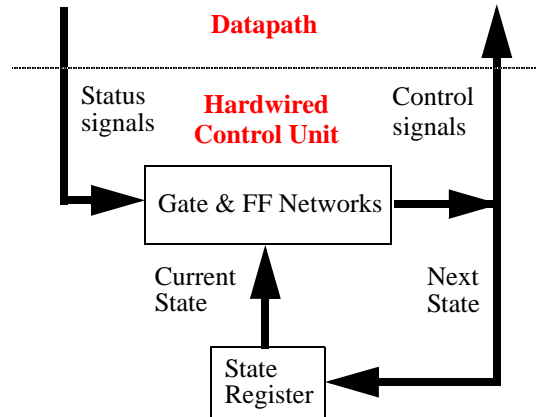


2.3. The Control Unit Design

Now, we focus on the control unit. The control unit controls the data unit. It determines which microoperation happens when. It also determines which datapath components are involved for that microoperation. The control unit has status signals as inputs and control signals as outputs. The circuit in the control unit is called “**sequencer**.” It is the one that determines the sequence of microoperations, hence the name, sequencer. The sequencer goes through steps, known as **states**. The sequencer may “jump over” steps also. But, it knows exactly which step it is in by using a **state register**. That is, the state register indicates the state the digital system is in at any moment. For example, if the state is 24 at the moment, then the value of the state register is 24. Based on the state and status signals, control signals are generated for the data unit and the next (state) value for the state register. The sequencer circuit is often an “**irregular**” circuit whose design is quite complex. The irregularity is due to the large amount of **random logic** (gate networks and flip-flops).

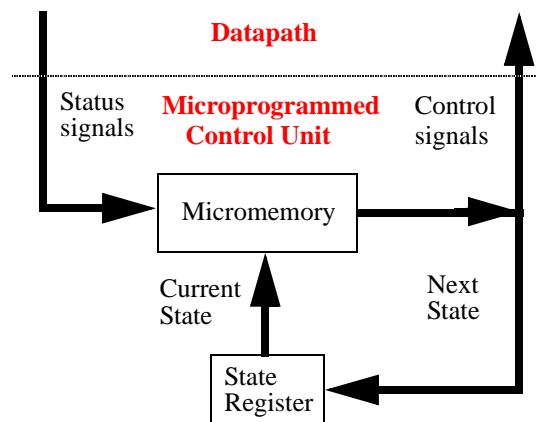


The sequencer is implemented by using **hardwiring** or **microprogramming**. Hardwiring generates control signals by **networks of gates and flip-flops** : for each control signal, there is a network of gates and flip-flops. Networks are needed also to generate next state signals. Overall, **hardwiring** leads to a **tremendous** number of gates. Since networks of gates and flip-flops are **random logic**, their development is very time consuming. The design, test, modify, manufacture and upgrade of huge amounts of random logic is monumentally difficult :



The advantage of hardwiring is that it is faster than microprogramming. If one wants the fastest possible sequencer for a digital system, hardwiring is used. Hardwiring is also more expensive. In the early days of computing, only hardwiring was known and used until microprogramming became practical in the 1960s. In the 1970s, hardwiring became more of a choice of supercomputing, as machine language instruction sets became complex. Only microprogramming was able to handle the complexity for other types of computers.

A microprogrammed sequencer generates control signals by using a special memory, a control memory, in the control unit, with little accompanying random logic as shown below. Microprogramming simplifies the control unit development. The control memory is stored bits representing values of the control signals for each state. There are also bits indicating what the next state is. The content of the control memory is referred to as control program or **microprogram**. Each location of the control memory keeps a **microinstruction** :



The control memory can be read-only, meaning the microprogram is never changed. If the control memory is a read-write memory, it can be changed, even at run time ! We, then, develop one microprogram for one machine language instruction set, have several microprograms in the main memory and download one of them to the control memory for a machine language instruction set that matches the current application.

As mentioned above, hardwiring is faster than microprogramming, but more expensive ! Microprogramming was

heavily used to implement sequencers in the 1970s and 1980s when most instruction sets were complex, CISC. In fact, microprogramming stimulated complex machine language instruction set design to reduce the semantic gap between high-level languages and machine languages. One of the most complex instruction sets is the DEC VAX-11/780 of the early 1980s, with more than 300 instructions. It was a super-mini computer, i.e. not even a mainframe.

With the acceptance of the RISC idea in the late 1980s, hardwiring was used since RISC instruction sets were simple. However, in the 1990s, as computer sales were increasing, leading to fierce competition and Moore's law holding, designers started to mix hardwiring and microprogramming. They use one of the concepts discussed earlier : [make the common case fast](#). Hence, they use hardwiring for frequently used instructions and microprogramming for other less frequently use instructions which are often complex instructions ! The MIPS machine language instruction set is a simple instruction set, a RISC set. It is acceptable to use hardwiring. Microprogramming can also be used. The hardwired and microprogrammed versions of the EMY CPU control unit are designed in class with a supporting handout.

We conclude the Digital System Design section by listing three seemingly unimportant points about it. When we deal with digital systems, we need to remember that

- i) We cannot short circuit outputs of gates and flip-flops unless they are tri-state.
- ii) Registers take on new values in the clock period after their write control signal is turned on
- iii) Buses take on new values in the clock period their connection signal is turned on

3. CPU Design

A Central Processing Unit, CPU, is a digital system. The CPU digital system performs microoperations to run machine language instructions. It interacts with other digital systems which are the memory and I/O controllers for correct operations. The CPU design follows digital system design principles discussed in the previous section. That is, the first partitioning of the CPU leads to a datapath (data unit) and a control unit (sequencer). The datapath is further divided into smaller blocks. Most important components in the datapath are registers, ALUs and buses. The control unit determines the sequence of the microoperations. It is implemented either as hardwired or microprogrammed or both (hybrid) where the hardwired implementation results in the fastest execution.

3.1. CPU Design Details

One can develop a CPU by simply following the steps in the previous section. However, coupling the development to the computer levels can result in a better design. That is, the CPU development can be done in the context of computer levels, from the applications through the transistor level. Below, we give a number of points that relate the CPU design to the architecture and digital logic levels.

To design a CPU we follow steps I through V below :

I) The design of the CPU is started after the architecture of the computer is designed, that is after the machine language instruction set of the computer has been determined.

II) On the organization level, main memory requirements for communication with the CPU are studied. These include the characteristics of lines between the CPU and the main memory : control lines and data and address buses.

III) The CPU is designed to execute for a selected group of instructions. Afterwards, the CPU is incrementally modified for each remaining instruction, until all the instructions are executable by the CPU.

a) The CPU is designed for the selected group of instructions, by going through the following steps :

i) the architectural operation (semantics) of one of the machine language instructions is analyzed,

ii) the instruction format of the machine language instruction is analyzed,

iii) by using the instruction format, the architectural operation is converted to a number of microoperations. These microoperations are then distributed to a number of states, resulting in the first version of the high-level state diagram of the CPU. But, what are these microoperations ? Some of them are required for all instructions and the remaining ones are instruction specific.

First, we must realize that the CPU runs a machine language instruction by going through a number of major cycles, containing microoperations. A major cycle is at least one clock period long. Second, for every instruction, the first two major cycles are the same for any CPU, including the EMY CPU. The remaining major cycles depend on the instruction run at the moment and the CPU designed.

The major cycles of the EMY CPU are as follows :

- The **fetch** cycle, a mandatory major cycle for all instructions :
 - fetch the instruction from the memory,
 - update the program counter.
- The **decode** cycle, a mandatory major cycle for all instructions :
 - the instruction just fetched is tested (decoded) to determine what it is.
 - the EMY CPU also prepares operands, in case they are needed by the instruction later.
- The **execute** cycle, a major cycle for all CPUs, but some CPUs have it later in the run :
 - a number of microoperations, specific to the instruction, are performed.
- The **memory** access/R-type completion cycle (**not** needed by control instructions of EMY) :
 - a memory data access or an A/L result write to a GPR is done.
- The **write back** cycle (**not** needed by R-type and control instructions of EMY) :
 - the data read from the memory is “written back” to a GPR.

iv) The high-level state diagram is worked on to execute the remaining instructions in the selected group one-by-one. For that, we repeat the three steps above for each instruction.

v) Once we have the high-level state diagram for the selected group of instructions, we obtain the datapath of the CPU from the high-level state diagram. This includes determining the buses and which register is on which bus. We can obtain the high-level state diagram in terms of buses, once we have a working high-level state diagram and a working datapath for selected group of instructions.

vi) We develop the low-level state diagram of the CPU by using both the high-level state diagram in terms of buses and the datapath : for each state we determine which control signals must be 1 and which ones must be 0 to accomplish the microoperations of that state.

viii) Finally, the control unit of the CPU is designed :

- If the CPU is to be hardwired, Boolean expressions are derived from the low-level state diagram then converted to the NOT/AND/OR circuits. If necessary, flip-flop circuits are attached.
- If the CPU is to be microprogrammed, the low-level state diagram is converted to a table with 1s and 0s. Then, the table is stored on the micromemory.

b) The CPU is modified to execute the remaining instructions one-by-one. For that, we repeat the seven steps below for each remaining instruction :

i) The architectural operation of the machine language instruction is analyzed,

ii) The instruction format of the machine language instruction is analyzed,

iii) The current high-level state diagram is examined to help us prepare the list of microoperations needed to run the machine language instruction. Then, we distribute the microoperations to the existing states on the high-level state diagram and to **possibly new states**. While we do that it is important to remember that the machine language instructions that were executable before, must still be executable after the modification of the high-level state diagram.

iv) Simultaneously, the datapath of the CPU is modified according to the modified high-level state diagram. We may add new buses, registers, MUXes, functional units in the ALU, new wires, new gates and use larger MUXes. Again, it is very important that all instructions executable before, must be still executable after the datapath is modified.

v) We modify the high-level state diagram in terms of buses.

vi) We modify the low-level state diagram of the CPU by using both the high-level state diagram in terms of buses and the datapath.

vii) Finally, the control unit of the CPU is modified

The CPU designed above is a non-pipelined CPU. It executes only one instruction at a time. Only instruction is present in the CPU at a time. Today's CPUs are pipelined where multiple instructions are executed simultaneously. The CPU has to keep track of instructions which are in different cycles of their execution. In addition, due to pipelining, the clock period is very short. Thus, the CPU design is different. The CPU now consists of a number of stages. Each stage corresponds to one cycle of execution. Each stage takes one clock period duration. The CPU is designed so that each stage is a separate digital system with its own control unit. Then, the CPU consists of as many digital systems as there are stages.

Modern microprocessors have multiple cache memories, such as Level 1 cache, Level 2 cache, etc. besides a CPU on the chip. These caches are separate digital systems that interact with each other and the CPU. The caches have their own control units. In today's terminology, if there is a single CPU on the microprocessor chip, then there is a single core on the chip. If there are multiple cores, then there are multiple CPUs each one of which is pipelined. To save space and power, the pipelining of CPUs on multicore chips is simpler than the pipelining of a single core on a chip. The simplicity is that there are less number of stages, less hardware to search for speeding up the execution and the clock period is longer. Finally, in multicore systems, there are shared cache memories whose design is more complex than non-shared cache memories in single-core systems. The complexity is to guarantee that a core that needs data gets the most up-to-date value of the data that may be generated by another core. This is what is called cache coherency. There is also cache consistency which further complicates the design of these shared cache memories.

3.2. Implementing the EMY GPR Registers

The EMY integer GPR register set in the CPU is implemented such that two register reads and one register write can be performed in parallel in one clock period. Thus, the GPR set has two read ports and one write port. The GPR register set is implemented as shown below :

