Module on

Computer Architecture

for

Foundations of Computer Science

by

William J. Taffe

This page is intentionally blank.

This version was modified on Monday, October 01, 2001 at 1:45 PM

Computer Architecture is the discipline of designing computer systems. Just as the architect of a building must visualize the overall design and conceptualize a structure which meets the goals of the building (a house, an office building, a prison), a computer architect must develop the overall structural plan for a new computer system. The architect must develop plans for the engineers who will eventually carry out the details of the design in electronic hardware and must make the appropriate types and level of drawings for the engineers to follow. Some of the drawings look like blueprints with boxes and wires, others are more mathematical. The architect must be able to envision the "big picture" but not at the expense of creating an unbuildable structure.

In this module, we will look at some of the aspects of a computer comprise the computer's architecture. We should also recognize that within computing the word architecture is sometimes used to characterize the overall structure of other aspects of the system such as software. Software engineers sometimes talk about software architecture.

We can roughly characterize a computer's architecture as having several "building blocks":
- Central Processing Unit (CPU)
- Memory (Main memory and cache)
- Mass storage (disks, CDROM, tapes, etc.)
- Input/Output systems (keyboard, monitor, printer, etc.)
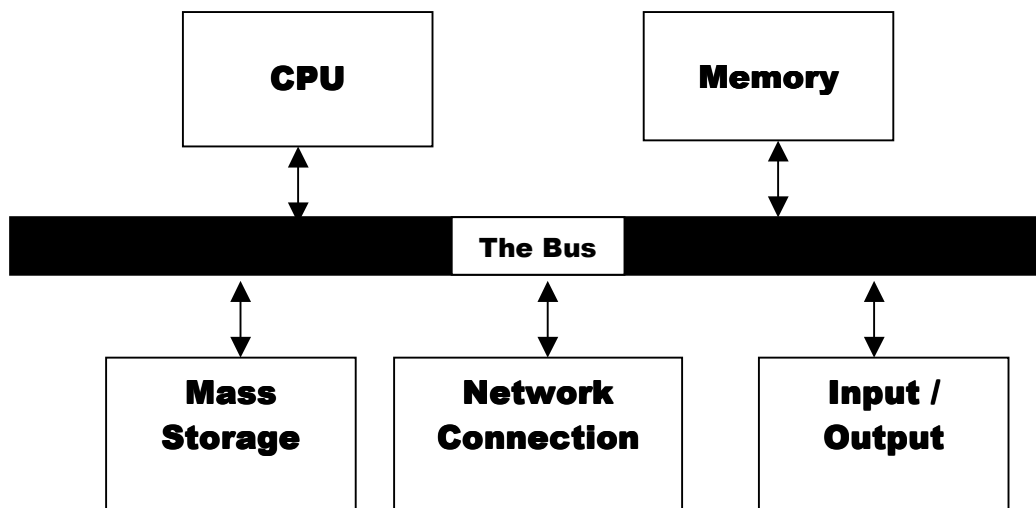- Network connections
- Bus or other interconnectivity

Figure 1

Each of the basic modules communicates with the others through the bus, which is simply a collection of wires. A bus is often characterized by its width (for example 32 bits wide, or 64 bits wide), or by its throughput, the number of bits per second it can transmit (for ex.100 Mbps). Naturally wider and faster buses are more expensive and all other things being equal a wider bus has a larger throughput.

Each of these basic modules in turn can be broken down into smaller sub-units. We'll examine some of these sub-units in the following sections.

The CPU

The Central Processing Unit is what is often called the chip. In past years, and for large systems today, the CPU consists of various IC's each of which has a specific function. However on systems such as the PC, the Mac, and even on many workstations, the CPU resides on one "chip". Chips have names such as the 80486, 86040, etc. which specifies the specific type of CPU. The common Pentium processor" originally developed by Intel is called the i80586 although other manufacturers also make the chip. Each of the chips has an internal clock (perhaps driven by an external clock) which simply means the number of beats, or "clock ticks" per second at which the chip runs. A Pentium chip might have a clock of 166 MHz (166 million beats per second), or 300 MHz or 400MHz or more. Figure 2 shows the typical clock speed of common PC chips (the i80x86 family) over the past decade.
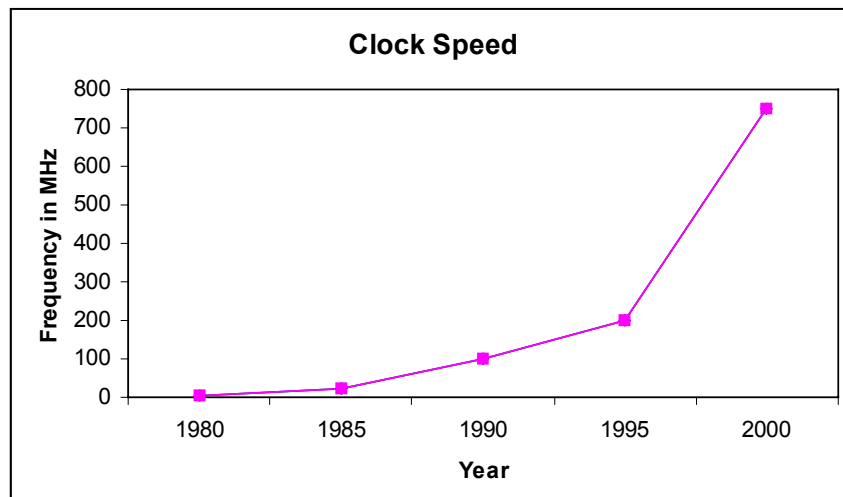
Clock Speed

Figure 2

Clock speed is one measure of how fast a chip is able to execute instructions but not the only measure. The internal design of the chip, the set of instructions it uses are also very important, the cache, the bus and all as important as clock rate. If all other aspects of a chip are equal, its relative speed can be measured by its clock rate, but usually other aspects of the architecture have a very strong influence. So clock speed is useful but not a determining factor in the power of the CPU.

The CPU itself has several subunits, linked by a bus internal to the CPU. Figure 3 is a schematic drawing of a typical CPU of the type known as a General Register Machine. The typical subunits are:
- The Registers
- The Control Unit
- The Arithmetic Logic Unit (ALU)
- The internal bus

The Registers A CPU typically has several registers: a group of registers collectively called, the Register File, the Instruction Register, the Program Counter (register) and other Special Purpose registers. A register is simply a storage location for one "piece" of information. Usually the "size" of a machine is stated in terms of the size of its registers. A 32 bit machine has registers that hold 32 bits or are "32 bits wide". The width of the registers is what determines the "word size" of the CPU. The word is the size of the machine's registers. Our registers can hold 5 characters. They are restricted to the digits (0 .. 9) and the minus sign (-). Therefore the largest number that can be represented (the most positive

number is `99999`. And the most negative number is `-9999`. If the minus sign is missing, the number is considered to be positive.

When a new instruction is fetched into the CPU (from the memory) it is put into the Instruction Register (IR). The role of the Instruction Register is to be the storage location for the instruction currently being executed.

The Program Counter, regardless of its name, doesn't count anything. It is the register that holds the address in memory of the next instruction to be executed. After each instruction is fetched into the CPU (and placed in the IR), the program counter register (PC) is incremented so that it holds the address of the instruction to be executed next. There are other "special purpose" registers lumped in this diagram in the box "Special Purpose Registers" which differ considerably depending on the machine. We won't discuss these further in this module. Finally the Register File, is simply a group of general purpose registers, usually 8, 16, or 32 registers (a nice power of 2), which are used for the temporary storage of data that is brought in from the memory and is about to be processed. Each of the registers is connected to the internal bus.
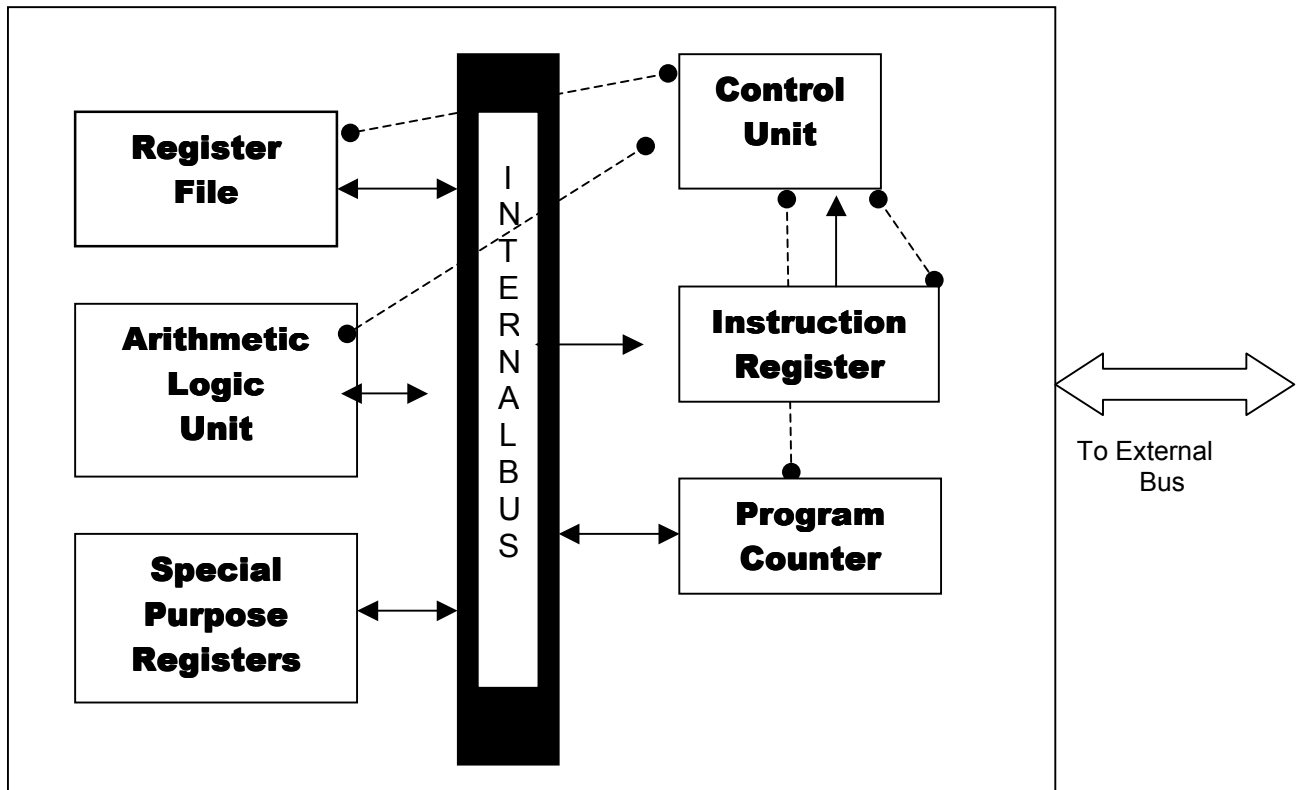

Figure 3: The CPU

The Control Unit (CU) The Control Unit is the "brains" of the CPU. It receives information from the Instruction Register (arrow from IR to CU) which it "interprets" by its electronic circuitry to determine what series of actions should be taken to execute the instruction. It then issues a series of "control signals" via wires to each of the other units of the CPU (the dashed lines with rounded heads ●-------●) which makes them take the proper actions to carry out the action of the instruction in the IR. After completing the series of actions, the CU sends a control signal to the PC and the memory. This causes the PC to send its value to the memory as an "address" and the memory to send the data at the address sent by the PC to the CPU. The CPU interprets this new data as an instruction and stores it temporarily in the IR

(overwriting the value that was in the IR). This is the basic cycle of activity in a CPU and is known as the fetch/execute cycle. The fetch/execute cycle is controlled by the Control Unit.

The Arithmetic Logic Unit (ALU)   The Arithmetic Logic Unit, as its name implies, does the arithmetic and logic calculations inside the CPU. All of the addition, subtraction, multiplication, division, ANDing, ORing, NOTing, and similar functions are done by the ALU. The ALU gets the data to be operated on from the Register File, and is sent a Control Signal by the Control Unit telling it what type of operation to perform. It sends its results back to one of the registers in the Register File.

The Fetch/Execute Cycle   The fetch/execute cycle was described briefly above and will be described here in more detail. It is one of the most fundamental aspects of the CPU.

Fetch
- The number currently in the Program Counter is sent to the Memory
- The memory interprets that number as an address
- The memory sends the data stored at that address in memory to the CPU
- The CPU interprets that data as an instruction and puts it in the Instruction Register

Execute
- The data in the PC is incremented so that it contains the address of the instruction after the one just fetched.
- The data in the IR is examined by the Control Unit (this is called "decoding")
- The control unit sends signals (in sequence) to the other parts of the CPU which causes them to do the things necessary to execute the instruction.
- The control unit begins the next fetch

This is illustrated below in Figure 4.

Fetch

1. **PC data -> Memory**
2. **Memory Data -> IR**
3. **Control Unit goes into Execute Phase**

Execute

1. **PC is incremented**
2. **CU examines IR**
3. **CU sends control signals**
4. **Other units perform their appropriate tasks**
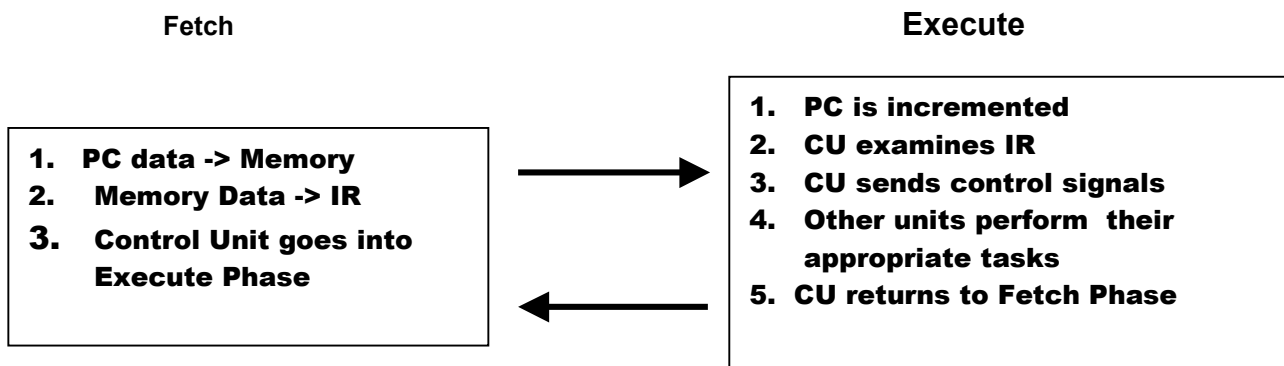5. **CU returns to Fetch Phase**

Figure 4: Fetch/Execute Cycle

The Machine Instruction   The machine instruction is simply a code, or a binary number (however you prefer to think of it) which contains all the information necessary for the control unit to send the correct control signals to the other parts of the CPU. There are many ways to design machine instructions and the design is often called the Instruction Set Architecture (ISA). In an introductory module, we cannot describe every type of instruction set architecture , but instead will concentrate on one type, the load/store structure in a type of Architecture known as a Reduced Instruction Set Computer or

RISC. We will illustrate the features of the instruction set by introducing a small simulated computer the DRC, the Decimal RISC Computer.   For ease of comprehension, this computer presents everything in base 10, although if it were built in hardware it would use base 2. The ISA of this computer is sufficient to write real programs which can be entered into the simulator and executed.

A machine instruction has two main tasks to perform. First it must tell the CPU <u>what</u> to do, and secondly, it must tell the CPU **where** to get the data from. The "what" is often called the <u>opcode</u>, and the "where" is often called the <u>operand</u>. There are three basic types of machine instructions:
1. Instructions that move data from one place to another (load/store or move instructions)
2. Instructions that do arithmetic (or logical operations) on data (ALU instructions)
3. Instructions that control what instruction to execute next (control instructions).

In this computer, all instructions consist of 5 characters … which can be 5 digits (0..9) or 4 digits and the minus sign (if the minus sign is missing the + sign is assumed.). The first 2 digits are the opcode or the instruction's number.   The last 3 digits are the operand. Since there are 2 digits for the opcode, the opcode can range from 00 to 99. Therefore this computer could have 100 instructions. But it doesn't; it has far fewer because they aren't necessary and one of the design characteristics of a RISC architecture is to include only the instructions that are necessary, and not to create new instructions that aren't needed, and might be used very infrequently.

Some instructions might not have an operand (for example the instruction that tells the CPU to STOP.) Others have 1, 2 or 3 operands. We'll learn more about these through examples.

Machine Code and Assembly Language

In addition to their opcode, all instructions have a <u>mnemonic name</u>. This is a word that sounds like what the instruction does, and makes it easier for the programmer to remember the instructions. For example, the store instruction is instruction number 5 and it's mnemonic name is STM. All names are 3 characters long. When we write programs using the mnemonic names we say the program is written in <u>Assembly Language</u>. When we write programs using the instruction numbers, we say the program is written in <u>Machine</u> <u>Language</u>.   A program written in Assembly Language is easier to write, find and fix errors that we might create, and is easier for humans to read. However, the computer cannot read assembly language, and so all programs written in Assembly Language must be translated into Machine Language before the computer can execute them. We call this process <u>assembling</u> a program. It can be done by hand (that's the way we'll do it here), or we can write a program to do it. If it is done by a program, that program is called an <u>assembler</u>. We'll first study the Instruction Set (or Language) of DRC1 through its representation in Assembly Language.

Load/Store Instructions

One of the characteristics of the load/store ISA, is that it separates the activities of the ALU and memory. Instructions that do arithmetic don't interact with memory. And instructions that work with memory don't do arithmetic. In a load/store architecture, data is loaded from memory into the general purpose registers. ALU instructions manipulate the data in the general purpose registers. And load/store instructions put the altered data back in memory.   When data is moved from the memory to the register file, it is called a <u>load</u>. And when data is moved from the register file to the memory, it is called a <u>store</u>.

There are only 5 instructions in the load/store group.

In a load store instruction, the opcode tells whether it is a load or store, and also the <u>addressing mode</u>. If we want to load some data, we need to know where the data is, and where to put it. In a load/store computer we always put the data in a register, but there are several different ways to specify the address of the data, or where the data is coming from. The names of these addressing modes are:
- Immediate addressing
- Memory Direct addressing (sometimes just called direct addressing)
- Register Indirect addressing (sometimes just called register addressing)

In immediate addressing the data is contained in the operand. The data is part of the instruction itself. In Memory Direct addressing, the operand gives the memory address of the data. In Register Indirect addressing, the operand specifies a register that contains the address of the data. This last may sound confusing, but you get used to it quickly, and will find that it is a very useful, and necessary, type of addressing. The first three instructions, all of them load instructions, use these three types of addressing.

Instruction #00 is the LoaD Immediate (LDI) instruction The operand has the data, and the instruction simply puts the data directly into a register. As an example, `LDI R5, 60` means put the number 60 into register 5. This can be written in a mathematical-like notation known as Register-Transfer-Notation (RTN) as R5 ← 60. The number **must be positive** and between 00 and 99 (00 .. 99). Although, in general, data can have any value between -9999 and 10000 (-9999 .. 99999) in a LDI instruction we are limited to positive values that have only 2 digits.

Other examples:
```
        LDI R3, 45          R3 ← 45 (The number 45 goes into Register 3)
        LDI R9, 00          R9 ← 00 (The number 00 goes into Register 9)
```

At this point the observant student might ask, "What if I want to put a negative number into a register?" This takes 2 steps. First put the positive value of the number into the register and second make it negative (with an ALU operation that will be discussed later.) You might also ask, "what if I want to put a number larger than 99. This cannot be done through immediate mode and can only be done via the loader. Enter the number and its memory address in the Editor and when the Load button is pressed, it will be stored in memory. In the program you use the instruction LoaD Memory (LDM, to be discussed next) to load the number. If you want an explanation of the design alternatives that led to these decisions, ask your instructor.

Instruction #01 loads data from memory to a register by specifying the memory location that contains the data.   It is called LoaD Memory and its mnemonic is LDM.   The LDM instruction has two operands. The first specifies which register the data will be put into, and the second specifies which memory cell currently holds the data. For example: `LDM R5, M60` means load register 5 with the data that is currently in memory cell 60. This can be written in Register-Transfer-Notation (RTN) as   R5 ← M60.

Other examples:
```
        LDM R3, M45         R3 ← M45 (The data from Memory cell 45 goes into Register 3)
        LDM R9, M00         R9 ← M00 (The data from Memory cell 00 goes into Register 9)

        Or in general it can be written:
        LDM R, M            R ← M
                            where R is the Register number and M is the Memory Cell number
```

Note the difference between
```
        LDM R5, M60         R5 ← M60    and
        LDI R5, 60          R5 ← 60
```

The first puts the contents of Memory Cell 60 into register 5. The second puts the number 60 into register 5.

Instruction #02 loads data from memory to a register by specifying the register that contains the **address** of the data.   It is called LoaD Register and its mnemonic is LDR.   The LDR instruction has two operands. The first specifies which register the data will be put into, and the second specifies which register holds the address of the memory cell that holds the data. For example: `LDM R5,R7` means load register #5 with the data that is currently in memory cell whose **address** is in register #7. This can be written in RTN

as `R5 ← M(R7)`. We use (R7) after the M to indicate that R7 contains the memory address where the data should come from.

Other examples:

`LDR R3, R6`   `R3 ← M(R6)`   The data in R6 is a memory address.
The data is at that address.

Or in general instruction #02 can be written:

`LDR Ra, Rb`   `Ra ← M(Rb)`
where `Ra` is the Register number where the data will be put, Rb is the register number that contains the memory address and the parentheses around Rb indicate that we are using register indirect addressing.

This may seem weird, but it is something like a scavenger hunt. You might get a clue that says "go to a tin can behind a big tree in the center of the Town Common to find the next clue". You aren't given the clue but are told where to find it. In register indirect addressing, you aren't given the memory address, but are told where to go find it.

This type of addressing is very useful (almost indispensable) in processing loops that work with lists of numbers, a very common type of problem in computing.

So to summarize the three types of load instructions, if we had the data value of 15 stored in memory cell 20, and the memory address 20 stored in register 4, we could get the data value of 15 into register 1 by any of the following 3 instructions.

`LDI R1, 15`   or   `LDM R1, M20`   or   `LDR R1, R4`

Instructions #03 and #04.   There are no instructions numbered #03 or #04. We're saving these numbers for later versions of DRC1.

Instruction #05 stores data from a register into memory. It is called STore Memory and its mnemonic is STM. It uses the same addressing mode as the LDM instruction (2 operands) but the data moves in the opposite direction. For example,   `STR R5,M60`   means that the data in register 5 will be stored in memory cell 60. This can be written: `M60 ← R5`.

Other examples:
`STM R3, M45`   `M45 ← R3` (The data from Register 3 goes into Memory cell 45)
`STM R9, M00`   `M00 ← R9` (The data from Register 9 goes into Memory cell 00)

Or in general it can be written:

`STM R, M`   `M ← R`   where R is the Register number and M is the Memory Cell number

Note: the order of the operands does not indicate the direction of data movement. The name LDM or STM indicates the direction. LDM moves data from Memory to a Register. STM moves data from a Register to Memory.

<u>Instruction #06</u> stores data from a register into memory but like instruction #02 uses Register Indirect addressing to specify the memory location. It is called STore Register and its mnemonic is STR. It is very similar to the LDR instruction except that the data moves in the opposite direction. For example,

                        STR R5, R7

means that the data in register 5 will be stored in memory cell whose address is in register 7. This can be written: `M(R7) ← R5`.

Another example:
`STR R3,R1`   `M(R1) ← R3` (The data in R1 is a memory address. The data that is in register 3 will be stored at that address.

Or in general:

`LDM Ra, Rb`   `M(Rb) ← Ra`
                   where `Rb` is the Register number that holds the memory address where the data will be put, and Ra is the register that contains the data. The parentheses around Rb indicate that we are using register indirect addressing.
.

So to summarize the two types of Store instructions, if we had the data value of 15 in Register 1 and wanted to put it in memory cell 20, and the memory address 20 was stored in register 4, we could get the data value into memory by either:

        `STM R1, M20`
        `LDR R1, R4`

This completes the load/store instructions, instructions #5 and #6. Instruction #7 to #9 are reserved for future versions of the CPU.

<u>Arithmetic Instructions</u>

Now that you understand the ways to get data into the registers, we'll explore the operations that can be done on that data. In a load/store architecture, all arithmetic is done on data in registers, and the results are returned to a register.

The ALU (Arithmetic Logic Unit) ops (operations) lie have numbers between 10 and 29. Although this leaves instruction number space for 20 operations, the DRC1's ALU can only perform 8 operations so there's lots of "space" left over for a future, more powerful machine in the family.

<u>Instruction #10</u> is the Add instruction and its mnemonic is ADD. ADD adds the contents of two registers and puts the results in a 3rd register. `ADD R3, R4, R5` means add the contents of registers 4 and 5 and put the results in register 3 or in RTN: `R3 ← R4 + R5`.

Other examples:
        `ADD R5, R1, R2`   `R5 ← R1 + R2`
        `ADD R9, R3, R0`   `R9 ← R3 + R0`

This results of this instruction can contain an error however! If you add two numbers and the result is greater than the maximum number that a register can hold (99999) it creates an error which we can call an overflow.

<u>Overflow</u>

It is possible to make an error when executing an add instruction. If you add two numbers and the result is too big to put in a register … we have an <u>overflow.</u>   So an attempt to add two numbers like 55555 and 66666, each of which are legitimate numbers, and can be stored in a register, yield the result 111111 which is too big.

What should the CPU do? There are several design alternatives.
1.   The machine could give an error message and halt. This would be a pain but at least it the "truth would be known."
2.   The machine could put the 5 most significant digits in the register and continue … this could result in lots of wrong calculations with no warning to the user of the program.
3.   The machine could put the 5 least significant digits in the register and continue … this too could result in lots of erroneous results.
4.   The machine could put the 5 least significant digits into a register and send some kind of signal to the programmer. This is often done by what is called a called the status register. Since the maximum value of the digit (the most significant digit) that is lost is one ($99999 + 99999 = 199998$) the programmer would know the error. But this forces the programmer to check the overflow flag after each calculation.
5.   There are other more sophisticated solutions beyond the scope of this simple introduction.

At present, this computer, the DRC1, is designed to take the first alternative. Because the machine was designed as a learning tool, for learning how to a computer works and how to write programs in RISC-type assembly language, the design alternative to halt the program and explain the error was chosen. Whenever a number that can't be fit in a register is generated, the machine will halt with a message in the error window. The burden is on the programmer to be sure that numbers are always within range. One easy way to do this is to artificially restrict the use of data to positive values between 0000 and 9999 and negative values between -1 and -999 since (-9999) + (-9999) = -19998 which needs 6 digits for its representation.

Although "real" machines can handle bigger numbers, **all** computers have a finite word size, an inherent limitation on the size of numbers which can be formed. Some machines give no warning when that size is violated … and some programs on real machines have given some very strange and weird answers as a result!

The remaining ALU ops are very similar to the ADD instruction (with similar restrictions on size of numbers).

<u>Instruction #11</u> is the Subtract instruction with mnemonic SUB. . SUB takes the difference between two registers and puts the results in a 3rd register.   SUB R3, R4, R5   means subtract the contents of register 5 from register 4 and put the results in register 3, or in RTN: R3 ← R4 - R5.

Other examples:
            SUB R5, R1, R2   R5 ← R1 - R2
            SUB R9, R3, R0   R9 ← R3 - R0

The most common error in the SUB instruction is to reverse the registers … i.e. to think that
SUB R5, R1, R2   means R5 ← R2 - R1 instead of R5 ← R1 - R2.

Instruction #12 is the Increment (INC) instruction. It adds the value of "1" to the register specified:

```
INC R1              R1 ← R1 + 1
```

It is a very simple instruction, but useful. Incrementing is often done in a computational problem where we need to count the number of times something is done.

Instruction #13 is the Decrement (DEC) instruction, the opposite of the Increment instruction. It subtracts the value of "1" to the register specified:

```
DEC R1              R1 ← R1 - 1
```

It too is a simple, useful instruction. It is commonly used for looping, allowing us to "count down " in the loop.

Instruction #14 is the Negative (NEG) instruction. It puts the data that is in a register into another register with the opposite sign.

```
NEG R1,R2           R1 ← -R2
```

The two registers may be the same, i.e.   `NEG R1, R1`   This just changes the sign of a number in a register.

This completes the ALU ops. Instruction numbers #15 - #29 are reserved for future versions of DRC1. It is worth mentioning something about the kinds instructions that are not included, and why.

- Multiply and Divide. These can be done by repeated additions or subtractions in a loop. Multiplication will be demonstrated as an example. Division will be given as a homework problem!

- Logic instructions AND, OR, NOT … these instructions are very useful when dealing with binary numbers, but since we've decided to use decimal (base 10) in this simulator, they don't make much sense.

- Likewise the Shift operations (Shift Right, Shift Left and their variations) are less useful when dealing decimal numbers and so are not included in the ALU operations.

Jump and Branch Instructions

After each instruction is fetched, the Program Counter holds the address of the next instruction in sequence. To change this sequential order of instruction execution, we need instructions that change the "flow of control."   We need to be able to create a flow of control that can loop or that can branch to a different place in the program depending on the results of the previous calculation In assembly language there are two types of instruction which accomplish this change of execution, Jumps and Branches. With Jumps and Branches we can create the Loops and If-Then-Else type instructions available in higher-level languages.

A Jump instruction is "unconditional". A jump always happens when a jump instruction is encountered. A branch instruction is "conditional". The change in sequence might happen, or might not, depending on values stored in certain registers. Some people call branches "conditional jumps" and

some call jumps "unconditional branches". We'll just use "jump" for "it always happens", and branch for "it might happen depending on values stored in registers".

The execution of a jump instruction doesn't seem like it does much. The execution simply puts the address of the next instruction to be executed into the PC register. It doesn't execute the instruction, nor does it even fetch it. It simply "sets up" the PC with the correct memory address to fetch next. There are two jump instructions in DRC1, and they differ only by where the memory address is stored; whether it is included in the operand of the instruction (Memory Direct addressing), or whether it is in a register and the register number is part of the operand (Register Indirect addressing).

If a branch instruction, "fails", or "the branch isn't taken", or the condition isn't correct, the execution of the branch instruction **doesn't do anything**. We say it "falls through". There is no change in program control. The next instruction to be executed is the one that the PC currently holds. If the branch "is taken", or "is successful", then the instruction holds the information about memory address of the next instruction either in its opcode (Memory Direct addressing), or in a register that is specified in the opcode (Register Indirect addressing).

Instruction #30   This instruction is called JPM (JumP Memory) and specifies the next memory address as part of the instruction.

```
        JPM  M        PC ← #M
```

The use of the #M indicates that the value of the Memory location (M) is transferred to the PC, **not** the data stored at the location M.

Example:
```
        JPM 20        PC ← 20
```

Caution… this nmenonic is JPM (JumP Memory) not JMP.

Instruction #31   This instruction's mnemonic is JPR (JumP Register) is a type of indirect addressing and specifies the Register that holds the memory address that is to be transferred to the PC.

```
        JPR R          PC ←  R
```

The number that is in the register specified, is interpreted as the address of the next instruction to be executed and is transferred to the PC.

Example: If Register 5 held the value of 10

```
        JPR R5         would cause    PC ← 10
```


Instruction #35   is the first of the Branch instructions and has mnemonic BEQ (Branch if Equal to 0). The instruction is written:

```
        BEQ R, #M          if R = 0  then    PC ←   #M
```

Which means: "If the number in the register specified is equal to 0 then transfer the memory address to the Program counter." If it is not, **don't do anything**! This latter part is hard for some students to grasp. If the number in register is not the same 0, the instruction is over, and we go on to the next instruction in the program (the one pointed to by the PC).

```
Examples:
     BEQ R3, 40          if R3 = 0   then    PC  ←   40
     BEQ R9, 20          if R9 = 0   then    PC  ←   20
```

This instruction is often used to see if two data values are equal. By combining it with the subtraction instruction we can easily test this. If we subtract the two values, then the result will be zero. We then test on the register containing their difference to see if they were equal. It takes two steps but is fairly straightforward.

Instruction #36   is just the opposite of the previous instruction.   It is BNE (Branch if Not Equal to 0) and is written as:

```
     BNE  R,  #M    if R NOT = 0     then    PC  ←   #M
```

Examples:
```
     BNE R3, 10   if R4 NOT = 0   then    PC  ←   10
     BNE R9, 15   if R1 NOT = 0   then    PC  ←   15
```

If the number in the register specified is **not** 0, then put the memory address #M (given in the instruction) in the program counter. If the number in the register R is 0, don't do anything. This instruction is very useful for building loops in which we count down from some number to 0. We use the instruction to jump back to the top of the loop until we hit 0 when we "fall through" and continue with some other part of the calculation.

Both BEQ and BNE are not needed since any numbers that fail one pass the other and vice versa but they are both included for convenience.

Instruction #37   tests to see if a number in a register is greater than 0.
It is BGT (Branch if Greater than 0).   The instruction is written:

```
     BGT  R, #M        if R > 0    then    PC  ←   #M
```

Which says: "If the number in the register specified is greater than 0 then transfer the memory address specified to the Program counter." Otherwise, don't do anything!

Examples:
```
     BGT R3, 40       if R3 > 0   then    PC  ←   40
     BGT R9, 20       if R9 > 0   then    PC  ←   20
```

Instruction #38   tests to see if a number in a register is negative. It is BLT (Branch if Less Than 0) (no matter what it looks like it is **not** known as the Bacon, Lettuce and Tomato test!).   The instruction is written:

```
     BLT  R, #M        if R < 0    then    PC  ←   #M
```

Which says: "If the number in the register specified is greater than 0 then transfer the memory address specified to the Program counter." Otherwise, don't do anything!

Examples:
```
     BLT R3, 40       if R3 < 0   then    PC  ←   40
     BLT R9, 20       if R9 < 0   then    PC  ←   20
```

Part of the RISC philosophy is to provide the instructions absolutely needed plus any others that would be used very frequently. We don't need both BEQ and BNE, nor both of BGT and BLT. For example, if two numbers fail the BGT and BEQ tests, then they must pass the BLT test but they are used frequently so we provide this redundancy. However, there is no instruction for BGE (Branch if Greater than or Equal to 0) or BLE (Branch if Less than or Equal to 0) for the other similar comparisons. The ones given are more than sufficient. It is the programmer's responsibility to figure out what combination of instructions to use for any logical comparison of two numbers is necessary. This usually only requires a couple of instructions … and some careful thought.

We often want to compare two numbers. To do this, the second number is subtracted from the first and one of the Branch tests is applied. If we want to test for equality A = B, the result of the subtraction is checked for 0. If we want to determine if A > B we perform A - B and check to see if the result is greater than zero BGT, etc.

Control Instructions

The remaining two instructions both pertain to program control, but not in the same sense as the Jump and Branch instructions. They are the Stop (STP) and Pause (PAU) instructions. The Stop instruction halts execution and "turns off" the simulator. The Pause instruction halts execution but doesn't stop the simulator, rather it permits the user to push "run" again to continue the execution or to switch to the single step mode to continue execution. Most "real" computers have a Stop but the Pause is usually a "software breakpoint" or "hook to the operating system" that allows another program to analyze what's happening in a program. Since this computer has no operating system (other than you, the operator), the pause is just a way of giving you back the control.

Instruction #50 This is the Stop (STP) instruction **which must be used** to halt every program. If you don't tell a program to Stop, the program will continue into "never-never-land", try to fetch and execute nonsense, and "crash." The Stop instruction has no arguments and is written simply:

    STP

In DRC1, this just causes the program to stop.

Instruction #51 is Pause (PAU) and is more a suspension of the program than a halt. The program stops running and waits for user intervention in the simulator. It also takes no arguments and is written:

    PAU

Pause is also called a "breakpoint" which can be very useful in debugging a program. When a program runs well up to a certain point and then "goes nuts", you can put in a breakpoint right after the "good" section. In the simulator, you can do a Continuous Run until the breakpoint, and then switch to Single Step mode. This avoids having to press the Fetch and Execute buttons dozens of times to get to where the program crashes. If you haven't yet used the DRC1 simulator this might not make much sense. After you have used it, you'll have more appreciation for this instruction.

This finishes the 18 instructions of DRC1. Table 1 below summarizes this instruction set. The table also includes the "Machine Language" for the instructions. The concept of "machine language" will be presented after a group of examples of programs written in DRC1.

Example 1:

Problem: Add the values 10 and -15 and store the result in another location in memory at location 40. We can put explanatory comments into a program by preceding them with a semi-colon (;).

```
LDI R1, #10         ; Load the number 10 into register 1
LDI R2, #15         ; Load the number 15 into register 2
NEG R2, R2          ; Change +15 to a -15
ADD R3, R1, R2      ; Add registers 1 and 2 and put the result in register 3
STM R3, M40         ; Store the number in register 3 in Memory location 40
STP                 ; Stop
```

Note: Anything after the semi-colon (;) is a comment and is not part of the executable instruction.

Example 2:

Problem: Load two numbers into memory locations 30 and 31. Compare the two numbers and store the larger of the two numbers into memory location 40. If they're equal store one or the other.

For this problem we need a branch and therefore need to know where each instruction is located. The memory location of each instruction is given to its left.

```
00     LDM R1, M30         ; Load the first of the numbers into register 1
01     LDM R2, M31         ; Load the second number into register 2
02     SUB R3, R1, R2      ; Subtract the second from the first
03     BGT R3, M6          ; If the first is > the second branch to location 6
04     STM R2, M40         ; Otherwise store the second
05     JMP 07              ; Jump to the end
06     STM R1, M40         ; The first was largest, so store it
07     STP                 ; Stop execution
30     15                  ; Data for location 30
31     10                  ; Data for location 31
```

Symbolic Addresses

In writing assembly language progams, we often use "symbolic addresses" to symbolize a memory address. When we translate the program into machine language we change these symbolic addresses into real memory addresses.   Example 2 written with symbolic addresses It is shown below:

```
       LDM R1, M30
       LDM R2, M40
       SUB R3, R2, R1
       BGT R3, skip        ; use the symbolic address "skip"
       STM R2, M40
       JMP 07
skip:  STM R1, M40         ; to indicate this line
       STP
30     15
40     10
```

Three changes are immediately obvious. The first is that there are no memory location numbers except in the lines with the data.   Second, is that the BGT instruction has the symbolic address "skip" as its memory location. Third, is that the word "skip" is the first thing on the line with the STM R1, M40 instruction and is followed by a colon (:). The word "skip" is a symbolic address or symbolic location.

When this program is translated into machine language, we will put actual the memory locations into each line. When we get to "skip" we will look ahead to see what number will go in its location, and put in the proper number as was done in the example 2. There is nothing special about the word "skip". Any 4-letter word is OK. When it is the first thing on the line, it must be 4 letters followed by a colon; when it marks the symbolic location within a statement, it does not have a colon. From now on, we'll use symbolic addresses for all programs in assembly language.

Example 3:   A list of five numbers is in locations 30..34. Add the list of numbers using a loop and store the result in location 35.

```
        LDI R1, #30         ; load the number 30 into register 1, (data location)
        LDI R2, #5          ; load the number 5 in register 2 (data points left)
        LDI R3, #0          ; be sure to clear the register that will hold the sum
loop:   LDR R4, R1          ; load first data point into reg 4 (address in register 1)
        ADD R3, R3, R4      ; add it to register 3
        INC R1              ; increment register 1 to point to the next data point
        DEC R2              ; decrement the count of the numbers we still need to add
        BGT R2, loop        ; if count not down to zero branch to "loop" - do it again
        STM R3, M35         ; store the result in memory location 35
        STP                 ; stop
30      5                   ; data for location 30
31      -2                  ; data for location 31
32      17                  ; data for location 32
33      25                  ; data for location 33
34      -10                 ; data for location 34
```

Example 4:   There is a list of numbers in locations 20 - 25, some might be negative. Take the absolute value of each number and store the list of absolute values in locations 30 - 35.

```
        LDI R1, #20         ; this is the location of the data
        LDI R2, #40         ; this will be the storage location
        LDI R3, #6          ; this will be the "loop counter"
loop:   LDR R4, R1          ; load first data point into reg 4 (address in register 1)
        BGT R4, next        ; if its positive, skip the next step
        NEG R4, R4          ; change its sign … we're here only if its 0 or negative
next:   STR R4, R2          ; store the value
        INC R1              ; point to next data point
        INC R2              ; point to next storage location
        DEC R3              ; decrement the count of the numbers we still need to treat
        BGT R3, loop        ; if count not down to zero branch to "loop" - do it again
        STP                 ; stop
20      5                   ; data for location 20
21      -2                  ; data for location 21
22      0                   ; data for location 22
23      25                  ; data for location 23
24      -10                 ; data for location 24
25      4                   ; data for location 25
```

Machine Language

Earlier in this module, we described the machine language of the DRC1 but we've written all the programs in assembly language (which is much easier). To translate the programs into machine language, is a process called assembling the program. We could write a program to do this work or we can do it by hand. In order to learn what's involved, we'll do it by hand. It is a very straightforward process.

First we supply a memory location for each line of code, starting from location 00. This 2 digit number is the first thing on each line (as you saw in example 2). Secondly follows a space and then a 5-digit operation number. The first two digits are the instruction number (if it is number below 10 , you must write the leading 0, i.e. instruction 0 is written 00, etc.). This latter number depends on what type of addressing is used, immediate, memory direct or register indirect. This is summarized in the right-hand column of Table 1.

       R means a 1-digit register number
       RR means 2 1-digit register numbers
       RRR means 3 1-digit register numbers
       VV means a 2 digit data value
       MM means a 2 digit memory value
       X means it doesn't matter what you put, this location isn't used, but we usually put an X

For example the instruction `LDI R1, #10` would be coded as follows:

```
LDI R1, #10
     instruction number      00
     register number         1
     data value              10

     instruction code        00110
```

and the instruction `ADD R3, R1, R2` would be coded as:

```
     instruction number      10
     register number         3
     register number         1
     register number         2

     instruction code        10312
```

Data is stored with just the memory location and the data value, as will be seen in the examples. It is not necessary to "build" a five digit number. Just give the location and value, using a minus (-) sign if appropriate.

Example 5, contains the programs from examples 1 and 3 with the machine code lying to the right of the assembly language code.

Example 5


```
LDI R1, #10                    00 00110
LDI R2, #15                    01 00215
NEG R2, R2                     02 1422X
ADD R3, R1, R2                 03 10312
STM R3, M40                    04 05340
STP                            05 50XXX




       LDI R1, #30             00 00130
       LDI R2, #5              01 00205
       LDI R3, #0              02 00300
loop:  LDR R4, R1              03 0241X
       ADD R3, R3, R4          04 10334
       INC R1                  05 121XX
       DEC R2                  06 132XX
       BGT R2, loop            07 37203
       STM R3, M35             08 05335
       STP                     09 50XXX
30     5                       30 5
31     -2                      31 -2
32     17                      32 17
33     25                      33 25
34     -10                     34 -10
```


This completes the description of the DRC1 instruction set. Following table 1 is a set of 5 problems for practice with the simulator. A description of the simulator follows in Appendix 1 of this module. Unless you are an experienced assembly language programmer, it will probably be necessary for you to run the simulator, enter and execute the sample programs and do the exercises, in order to really understand this module.

DRC1
Assembly Language
and
Machine Language Summary

OP	Assembly	Meaning	Machine
00	LDI R, V	R <- #V	00RVV
01	LDM R, M	R <- M	01RMM
02	LDR Ra, Rb	Ra <- M(Rb)	02RRX
05	STM R, M	M <- R	05RMM
06	STR Ra, Rb	M(Rb) <- Ra	06RRX
10	ADD Ra, Rb, Rc	Ra <- Rb+Rc	10RRR
11	SUB Ra, Rb,Rc	Ra <- Rb-Rc	11RRR
12	INC R	R <- R+1	12RXX
13	DEC R	R <- R-1	13RXX
14	NEG Ra, Rb	Ra <- -Rb	14RRX
30	JPM M	PC <- #M	30MMX
31	JPR R	PC <- R	31RXX
35	BEQ R, M	iff (R = 0), PC <- #M	35RMM
36	BNE R, M	iff (R != 0), PC <- #M	36RMM
37	BGT R, M	iff (R > 0), PC <- #M	37RMM
38	BLT R, M	Iff (R < 0), PC <- #M	38RMM
50	STP	STOP (execute disabled)	50XXX
51	PAU	STOP (execute enabled)	51XXX

Table 1: DRC1 Instruction Set

Legend:
   #V or #M means the actual value of the number V or M
   M or R means the contents of Memory M or Register R
   M(R) mean the contents of the Memory location M who's address is contained in Register R
   X means "don't care" … you can put anything (including X)
   R is always 1 decimal digit. MM and VV are always 2 decimal digits

Appendix I

Documentation for

Decimal RISC Computer 1

The user interface to the DRC1 consists of two windows, the "Program Editor Window" and the "Computer Display Window". Each has a button that permits the user to return to the previous window.

Upon executing the program, the first window is the "Program Editor Window" shown below in figure 1. In this window, the user can enter a DRC1 program and its data. The program must be entered in DRC1 machine language. An example is shown below with 7 lines of code entered. Each line consists of two numbers. A 2-digit number specifying a memory location, and a number of up to 5 digits (including a "- sign" if necessary) that specifies a 5 digit operation or numerical data to be stored at that location. In the example below, the first 5 lines contain instructions that will be loaded into memory locations 00 -> 04 respectively, and two pieces of data (10 and -15) that will be put into locations 30 and 31. This program adds the numbers found in locations 30 and 31 and stores the result in location 40.
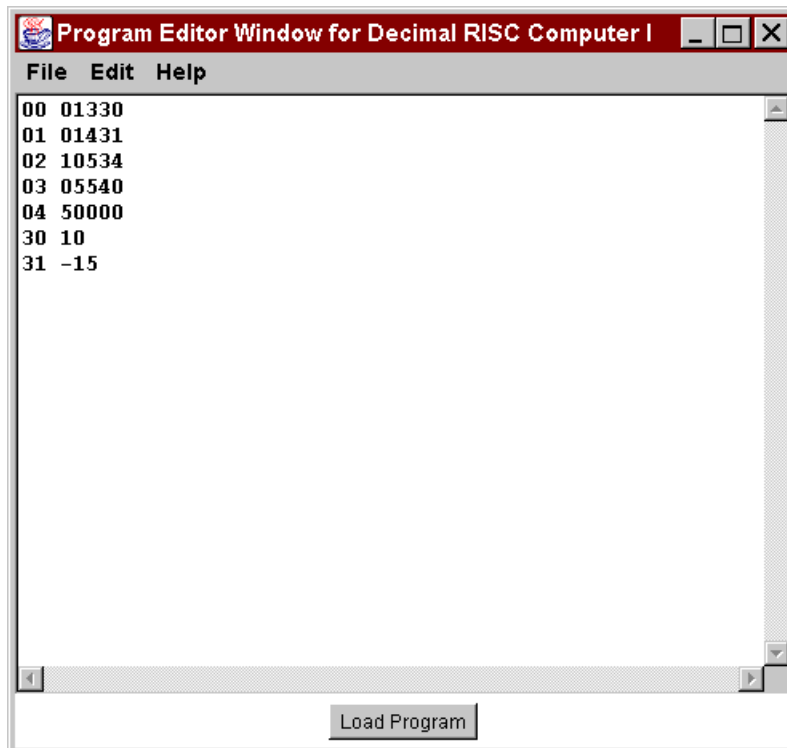

Figure 1

The Editor Window has a File menu with menu choices Open, Save, Save As, Print and Exit. All are typical Windows menu choices. It has an Edit menu with Cut, Copy and Paste. And a Help menu with a Help submenu (not yet implemented) and an About submenu giving information about the program version and author.

After entering a program in the manner described, the user presses the "Load Program" button found at the bottom of the window. The Editor Window disappears and the Computer Window opens with the program loaded into memory as shown in figure 2 on the next page.
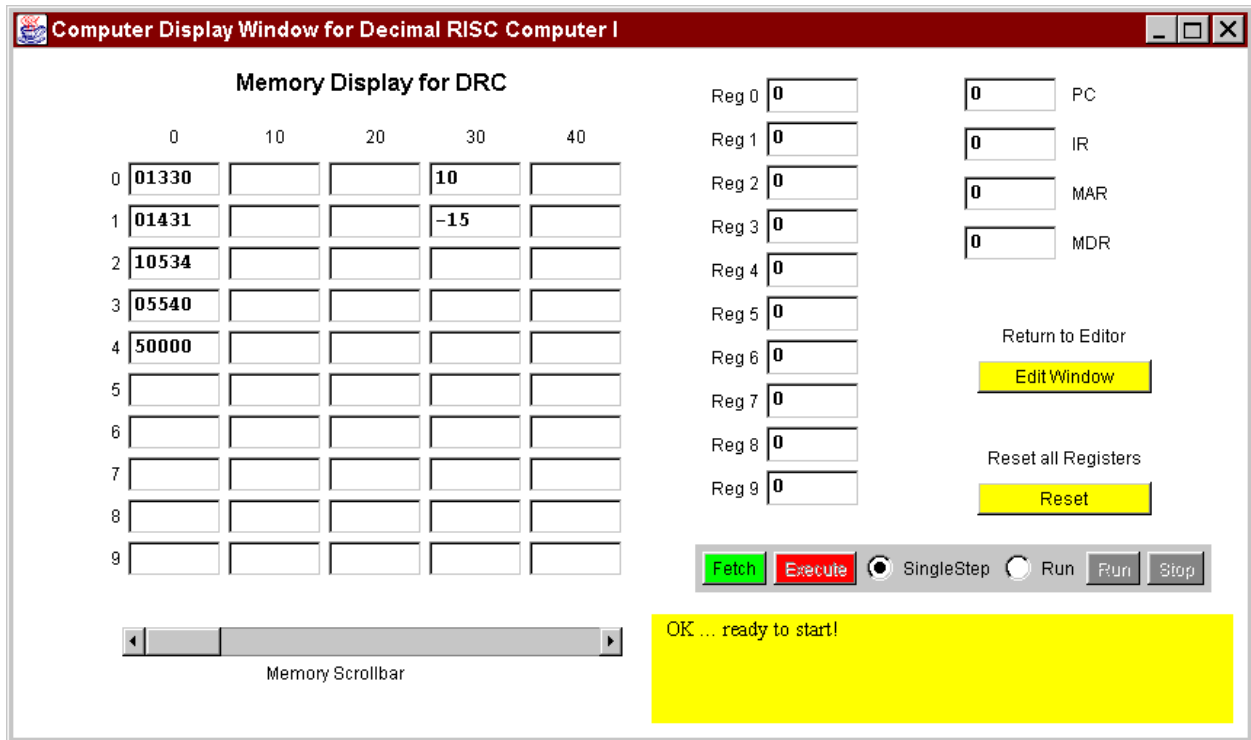
Figure 2

As is seen above, the Computer Window displays the Memory, the Register File, and the PC, IR, MAR and MDR registers. It has buttons to allow the user to reset all registers (to 0), and to return to the Editor Window. The Memory displays only 50 of the 100 memory locations at any one time, but the scroll bar at bottom permits scrolling through the entire memory. A panel of Control buttons lies below the Register File and Reset button, and a message screen lies below the Control buttons. The message screen provides user limited feedback throughout the execution of the program.

The program may be executed by using the buttons in the Control Panel. It may be executed in Single Step mode, or in Continuous Run Mode as selected by the radio button (Single Step - Run). In the Single Step mode the Run buttons are inactive and gray; the Fetch button is active and green and the Execute button is inactive and red. Green and red are used to symbolize active and inactive buttons in analogy with traffic lights. After Fetch is pressed, and the instruction pointed to by the Program Counter is fetched into the Instruction Register, the Fetch button turns red (and inactive) and the Execute Button becomes green (and active). In the Continuous Run mode, Fetch and Execute buttons are gray and inactive, the Run button green and active and the Stop button red and inactive. When Run is pressed the program executes until stopped by the software instructions Stop or Pause or when the Stop button is pressed. After pressing Run, the Run button becomes inactive and red and the Stop button is active and green.

If the program is halted by the Pause command (instruction 51), Run is activated and Stop inactivated. If it halts because of Stop (instruction 50), all buttons are inactivated (and turn gray) and the Single Step radio button must be selected in order to reactivate the execution. Note that this does not reset any registers. Register resets must be done manually by the Reset button whenever desired.

Whenever the user wishes to modify the program, or choose a new program to execute, the "Edit Window" may be pressed to return to the Editor Window (figure 1) where these changes can be made.