Designing the Computer

This is the most interesting part of the project. If you are familiar with assembly language programming, you know that each processor has a characteristic architecture. The architecture refers to the fixed characteristics that are built into the hardware. Elements of processor architecture include the size and nature of the instruction set, the size of the memory address space, the word size, the register structure, and the clock speed.

As with any engineering project, there are trade-offs in the elements that go into processor design. A processor with a large data width is more expensive to make, but it can process faster. The trend toward larger instruction sets and faster clock speeds seen in the 80x86 series is offset somewhat by RISC (reduced instruction set computer) processors that have smaller instruction sets, but perform each instruction more quickly. RISC computers may require longer programs, but with computer memory becoming very cheap this is not a severe hindrance.

The architecture of a hobbyist computer reflects the goals of the hobbyist. In general, a computer that works is the primary goal. High performance is very much secondary. A simple architecture reduces cost, and makes troubleshooting simpler. This increases the probability of success. Once the hobbyist has successfully built a simple processor, he may want to build another with higher performance. However, since building a working processor from scratch is a significant achievement, smaller is better.

I will describe the decisions I made in building my processor. In retrospect, I would have changed some things. However, this is only after I found that my simple processor worked, and worked well. My initial design was focused on successful completion, and not performance.

The most fun I had was designing the instruction set. Here, the hobbyist is completely free to make his own assembly language. I knew that the earliest computer, the Manchester Baby, operated with only seven instructions. I thought that it was reasonable to expand this to sixteen instructions. This meant that four bits of an instruction word would be taken up by the instruction code. The remainder of the instruction could be an address, or an operand for arithmetic or logical operations.

The address space of a hobbyist computer may be incredibly small by PC standards. In the 8-bit microprocessor systems I had built, I used only a hundred or so bytes for programs and data. I considered a total address space of 256 words for my processor, which can be encoded by 8 bits, but I opted for the more expansive 12-bit space of 4K words. This would be very ample for a hobbyist processor.

A 4-bit operation code and a 12-bit address space dictate a 16-bit instruction word size. This suggests an 8- or 16-bit wide memory. A 16-bit wide memory allows loading of an instruction in a single step. In retrospect, these choices of opcode, memory space and word size were optimal.

The next decision I made was an error. I decided that the internal data processing structure of the computer needed to be only 12 bits wide. This would be large enough for processing integers from 0 to 4K in a single step, and was large enough for address arithmetic. The program would be in ROM. The program memory would be 16 bits wide, but the data memory only needed to be 12 bits wide. This would save a little in complexity and cost. The instructions would cause 12-bit data to be fed to 12-bit registers, through a 12-bit ALU, and the results returned to 12-bit memory (or output on 12 or 8-bit devices.) The mistake, however, became evident toward the end of the project. After success seemed assured, I began to daydream of complex software running on my new machine. Then I realized by mistake: there was no provision for loading instructions from input to the RAM.

When the computer showed signs of life, I wanted to test it thoroughly. However, this meant putting even the numerous test programs in ROM, a tedious bit-by-bit process. I then modified my original design, by adding an extra 4 bits to the accumulator register and making RAM 16 bits wide. I eventually wrote a ROM program that allowed the processor to take serial character input from a terminal, translate this into 16-bit instructions (using a table for the upper 4 bits), and load the instructions in RAM. A character command would then shift execution to these instructions. The problem of "dumping" memory output to the terminal was more difficult, since there was no way of getting the upper 4 bits from the accumulator into the upper 4 bits of an 8-bit character output. I built a special purpose "byte-switcher" port, which would swap the upper and lower 8-bits of a word. All this could have been avoided by making the processor 16-bits wide throughout, which would not have been very difficult...in retrospect.

I wanted a minimal register structure, in part because a 4-bit opcode would not allow for complex register addressing schemes, and would also be easier to build. A simple register structure was also important because the part of the computer that contained the registers could not be tested fully by itself. It could only be tested when the whole processor was put together. I chose an accumulator-memory model, in which a main register, the accumulator, serves as the link between the processor and memory. The accumulator holds one operand of operations such as ADD, and receives the result. The processor also moves data between memory locations and between memory and input output by way of the accumulator. In addition to the accumulator, the processor would also need an instruction register, and a program address register (also called an instruction pointer, or program counter).

The minimum instruction set for a computer must contain some arithmetic or logical instructions, memory load and retrieval instructions, and program flow modifying instructions (jumps), at least one of which must be conditional. The minimal arithmetic instruction would be subtract, since this allows negation (subtract from zero) and addition (negation followed by subtraction). A clever programmer might be able to use the NAND logical operation to derive all the others, including subtract, since this is the root of all the other operations. However, with computer design, problems are solved by a combination of hardware and software. A full-function ALU is easy to design, and it seemed that 8 arithmetic/logical instructions (half of the allowed instruction set) would make a good project. I was confident that I could build such an ALU, and it could be tested completely on its own. The eight arithmetic/logical instructions I chose were add, add with carry, subtract, subtract with borrow, NOT, AND, OR, and XOR. I think this was a good choice.

The rest of the instruction set was available for memory access and flow control instructions. After some consideration, I decided to have only three memory access instructions. These were load accumulator immediate (the value is contained in the instruction itself), load accumulator from memory, and store accumulator to memory.

I now had five spaces in the instruction set left for program flow control. First, there is the simple unconditional jump. For conditional jumps, I chose jump on carry, jump on minus and jump on zero. The last spot in the instruction set was filled with an indirect jump instruction, that is, jump to the location stored in memory. This would allow some limited subroutine programming, using a memory cell to hold the return address.

In retrospect, it might have been better to have an indexed memory access instruction at the expense of one of the conditional jumps, or even the jump to memory instruction. However, indexing could still be done in a roundabout way by incrementing an instruction in RAM and jumping to it.

With the instruction set completed I now had a basic architecture from which I could draw detailed plans. The architecture described a processor that had sixteen instructions and operated on 12-bit data. It would have only one programmer-accessible register, the accumulator, and would have a 4K word address space. Instructions would be 16 bits wide, consisting of the 4-bit opcode and a 12-bit operand, which in most cases was an address. The exceptions were the load accumulator immediate instruction, in which the operand was a data value, and the NOT instruction, in which the operand was irrelevant. I somewhat arbitrarily assigned the following operation codes to the instruction set:

Opcode (binary)	Mnemonic	Operand	Description
0000	ADD	Address of operand	Adds the operand to the accumulator, stores the result in the accumulator
0001	ADC	Address of operand	Adds the operand and the carry bit to the accumulator, stores the result in the accumulator
0010	SUB	Address of operand	Subtracts the operand from the accumulator, stores the result in the accumulator
0011	SBC	Address of operand	Subtracts the operand and the complement of the carry bit from the accumulator, stores the result in the accumulator
0100	AND	Address of operand	Bitwise logical AND of the operand with the accumulator, stores the result in the accumulator
0101	OR	Address of operand	Bitwise logical OR of the operand with the accumulator, stores the result in the accumulator

0110	XOR	Address of operand	Bitwise logical XOR of the operand with the accumulator, stores the result in the accumulator
0111	NOT	None (ignored)	Logical bitwise inversion (complement) of the accumulator, stores the result in the accumulator
1000	LDI	Data	Loads the 12-bit operand from the instruction word into the accumulator
1001	LDM	Address of data to be loaded	Loads a 16-bit data word from memory into the accumulator
1010	STM	Address where data is to be stored	Stores the 16-bit data word in the accumulator into memory
1011	JMP	Target address to jump to	Transfers program control to the instruction in the target address
1100	JPI	Address containing the target address	Transfers program control to the instruction in the target address
1101	JPZ	Target address to jump to	Transfers program control to the instruction in the target address, if the 12-bit accumulator base = 0
1110	JPM	Target address	Transfers program control to the instruction in the target address, if bit 11 (leftmost bit of the 12-bit base) of the accumulator = 1
1111	JPC	Target address	Transfers program control to the instruction in the target address, if the carry bit = 1

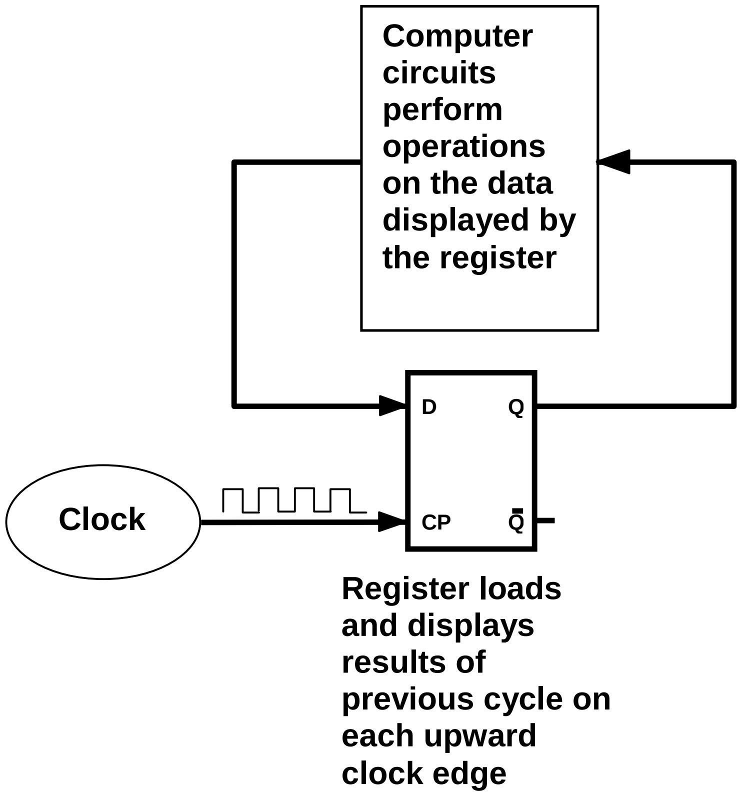
The instruction opcodes for the arithmetic-logical instructions are grouped together from 0000 to 0111. The lower three bits of these opcodes (000 to 111) will serve as a three-bit ALU opcode. This will simplify the control logic design later on.

The speed of the computer is dictated by the collective speed of the logic gates that need to change states with each cycle. The cycle time must be long enough to allow the slowest circuits time to finish operations. The registers, multiplexors and ALU have pathways of only a few to several dozen gates long. Assuming a gate time of 10 nanoseconds, a cycle time of several hundred nanoseconds would certainly be long enough. However, the computer memory is also part of the system, and this is usually the slowest component. I planned to use EPROM chips that had a 400 nanosecond delay between the request for the data and when it appeared on the outputs. In order to accommodate this delay, with time to spare, I chose a 1000 ns cycle time, which equals 1 MHz. Slow by modern standards, this is plenty fast enough to give the feel of true computing.

Building the Data Path

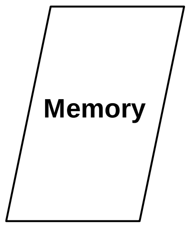
The data path of the computer is the ordered collection of registers that hold the data, the multiplexors that direct the flow of the data, and the ALU that operates on the data. The memory may also be considered part of the data path, although it is physically separated from the processor.

Recall the simple machine cycle diagrammed in the previous chapter.

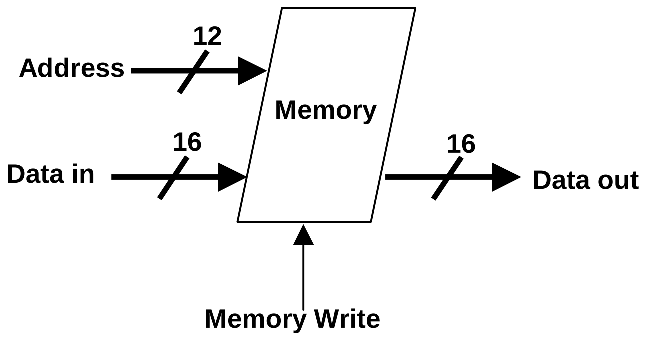

Computer circuits perform operations on the data displayed by the register
D
Q
Clock
CP
Q̄
Register loads and displays results of previous cycle on each upward clock edge

We will start at this point and diagram the processor data path, using as an example the execution of the ADD instruction. This instruction is executed by the processor in several steps, each taking exactly one clock cycle to perform. We assume for now that the processor is running. The mechanism to start the processor will be discussed later.

The ADD instruction that will be executed is part of a larger collection of instructions that reside in the computer memory. This collection of instructions is the program, and has been written and placed in the memory by the programmer. Exactly how this is done will also be discussed later. The computer memory is symbolized by a parallelogram.



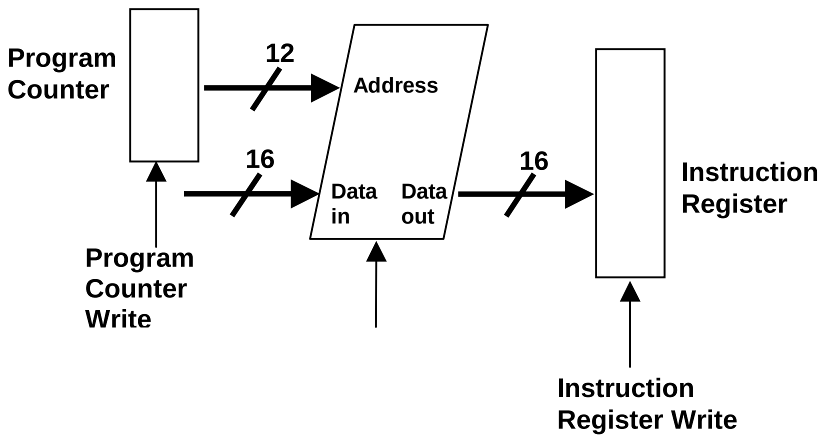
The memory in our computer will be 16 bits wide, and at present will be shown with distinct data inputs and outputs. The memory address is specified a 12 bit input.



We will assume for now that the memory acts like a large collection of registers in that a rising clock pulse edge at the Memory Write input will cause the data on the inputs to be loaded into the memory cell specified by the address. This data will then appear on the outputs. In the absence of a Memory Write pulse, the memory will simply display whatever data is present in the memory cell specified by the address input. The Memory Write signal will be derived from the processor control logic that will be described later. It is a single bit input.

The first step in executing the ADD instruction is to get the instruction from memory. This is commonly called the instruction fetch step. Since the processor is running, it contains in the Program Counter register the address of the ADD instruction we want to fetch and execute. To get the ADD instruction out of memory, we simply send the output of the Program Counter register to the memory address inputs.

After a few hundred nanoseconds, still well within the limits of one clock cycle, the desired instruction will appear on the memory data output lines. However, in order to execute the instruction we need to save it somewhere. We cannot simply keep it on the memory output lines, since we will need to get at least one of the ADD operands from the memory (the other operand is already in the accumulator register, left there by prior instructions). We will store the instruction in the Instruction Register.

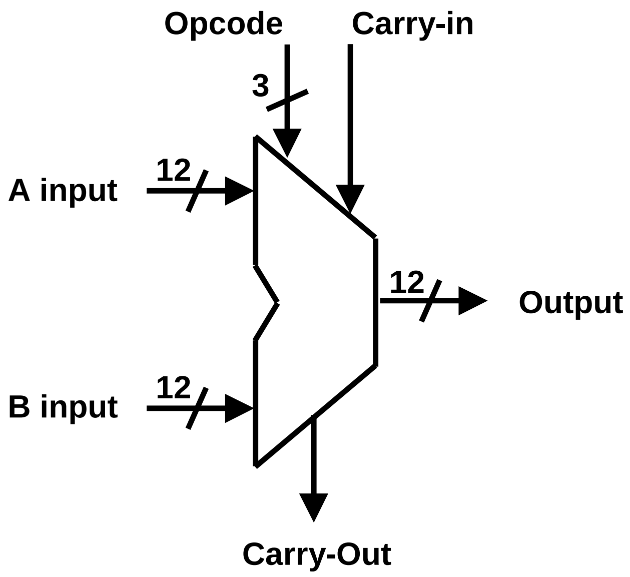


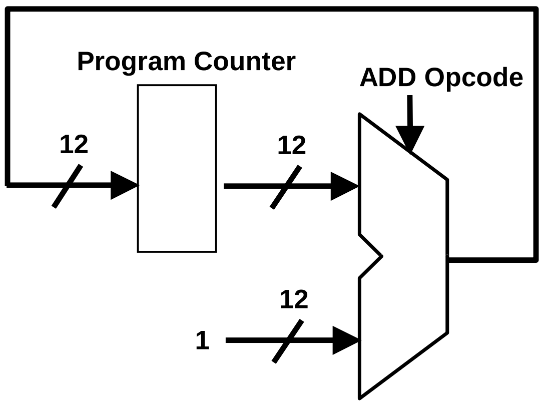
All of the registers, like the memory, have Write inputs. To complete the instruction fetch cycle, we need to send an Instruction Register Write pulse to the Instruction Register. This pulse, like all the other register write signals, will come from the control logic, to be designed later. We now have diagrammed all the parts of that data path needed to complete the instruction fetch.

After the instruction fetch, the computer will need to prepare itself to get the next instruction. Unless a jump instruction is being executed, the next instruction is in the memory cell immediately following the one just fetched. Therefore, we need to increase the value of the program counter by one. How can we do this?

The answer is simple. We will have an ALU built into the processor, and it has nothing to do in the first part of the instruction fetch. Why not use it to add 1 to the Program Counter? The ALU can do this while the memory is getting the current instruction. By the time the current instruction is ready to be clocked into the instruction register, the incremented program counter is ready to be clocked into the PC register. Remember, we are using rising clock edge triggered registers, and the increased Program Counter value will not appear on the Program Counter Register outputs until the program counter write pulse arrives. This will happen at exactly the same time that the instruction register write pulse arrives at the instruction register. There will not be enough time for the new program counter value to confuse the system with another instruction, since the instruction register will be closed for inputs long before the new instruction appears on the memory data output. Until we send another IR write pulse during the fetch phase of the following instruction execution, the current ADD instruction will abide securely in the Instruction Register.

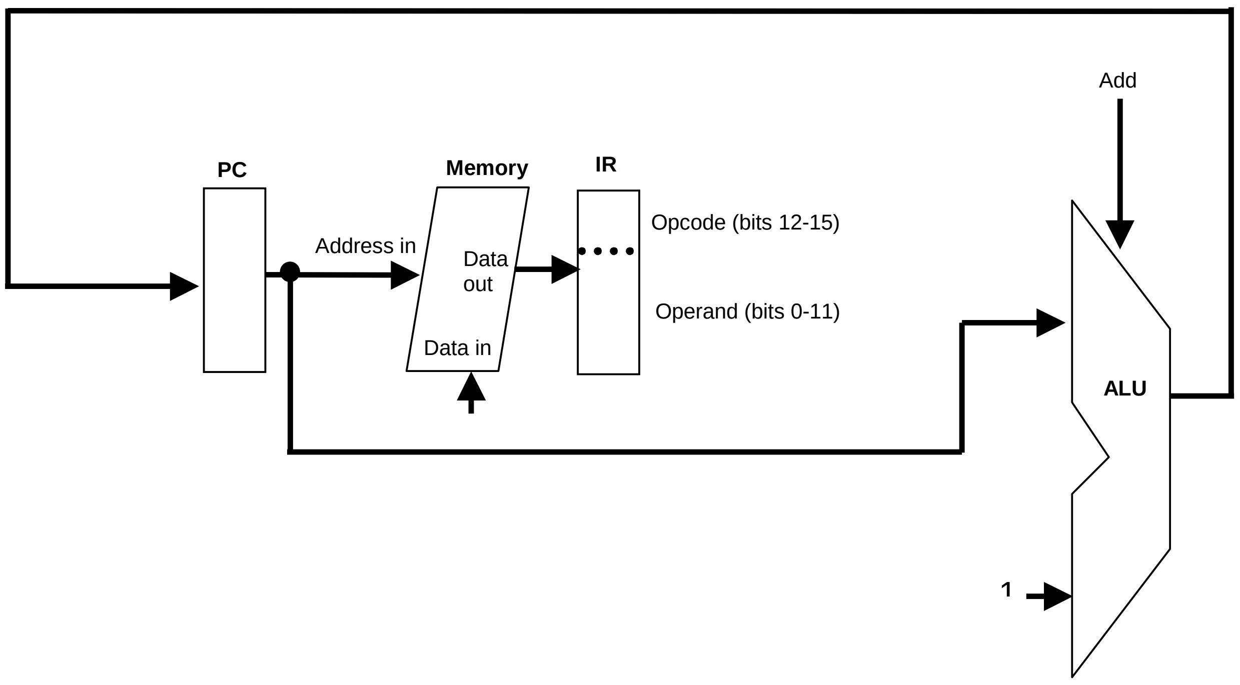
Here is a diagram showing how we can use the ALU to increment the Program Counter. Remember the diagram of the ALU from previous chapters. Our ALU will process 12 bits and have a three-bit opcode.





We do not use the carry in when the program counter is incremented. All we have to do is send the current Program Counter value to one input of the ALU, the 12-bit number 1 to the second (that is, binary 0000 0000 0001), and put the three bit ADD opcode on the ALU opcode inputs. After a short delay, the incremented Program Counter will appear at the ALU output and will be sent to the Program Counter inputs. It will wait there until the Program Counter Write pulse arrives, at which point the incremented program counter value will be loaded into the register, ready to fetch the next instruction.

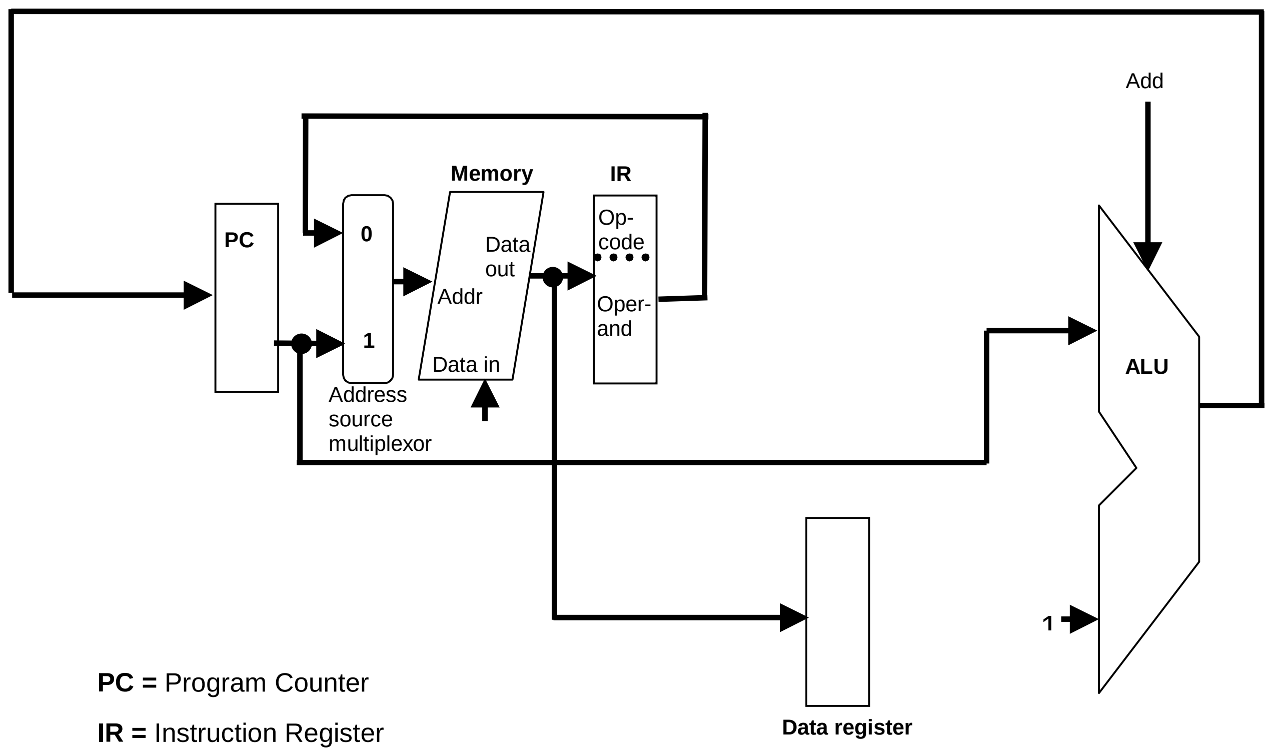
We have finished putting together the pathway for the instruction fetch and the program counter incrementation step. I will add to the diagram as we go through the rest of the ADD instruction execution. In order to make room in the diagram, I will not show the number of bits in each connection, nor the register write inputs from now on to reduce clutter.



At the end of the instruction fetch/program counter incrementation cycle, the Program Counter Write and Instruction Register Write inputs receive clock pulses. The rising edge of these clock pulses locks the results of this cycle into these registers at the same instant, and then the next cycle begins. In the diagram, I show that the upper four bits of the IR holds the instruction opcode, and the lower 12 bits the instruction operand. These two parts of the IR are sent on different paths, as I will show soon.

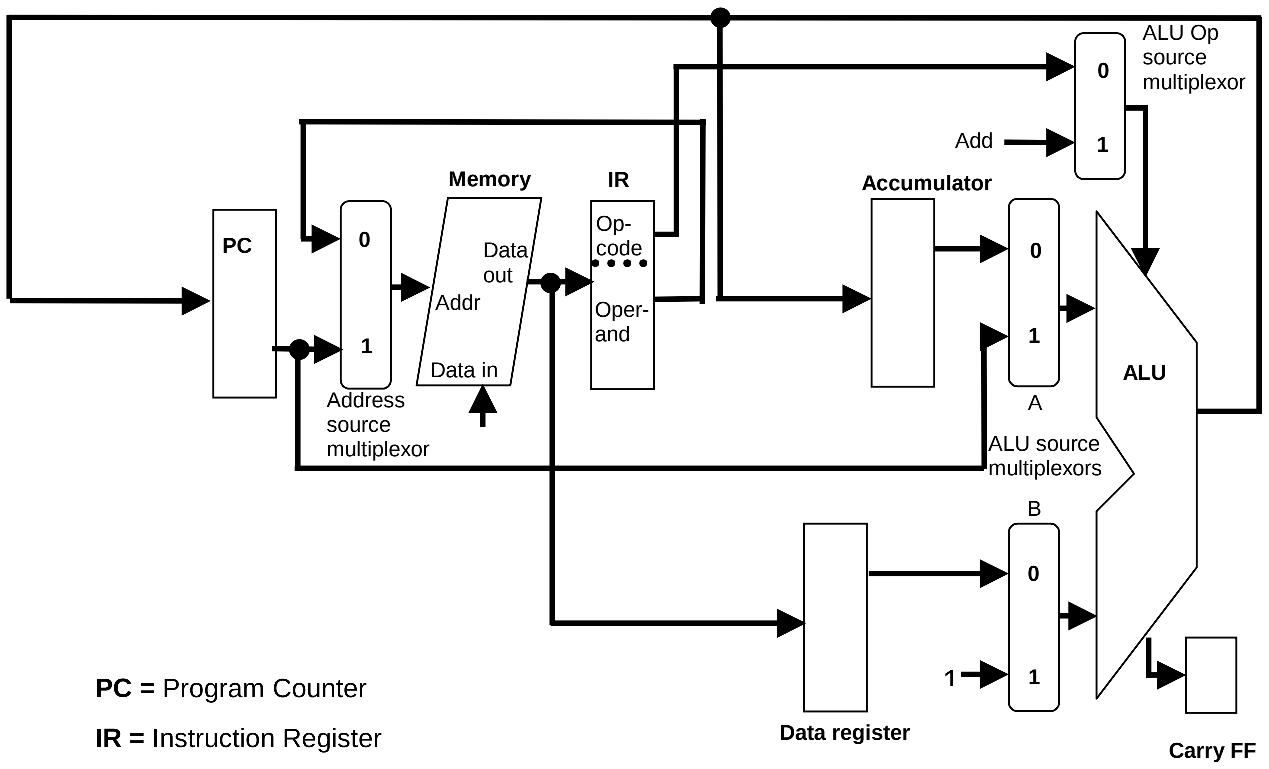
The next clock cycle is Instruction Interpretation. During this cycle, nothing happens in the data path. The instruction opcode (the upper 4 bits of the instruction register) is sent to the control logic. This opcode, along with the zero, minus, and carry condition flags, directs the control logic to set the control lines of the data path in such a way as to carry out the operation desired. In order to perform the ADD operation, the control logic will cause the data path to perform the following two steps, each taking one clock cycle. First, an operand stored in the memory will be fetched and placed in the Data register. Second, the Accumulator and Data register values will be sent to the ALU data inputs, and the three-bit ALU opcode for ADD will be sent to the ALU opcode input. At the end of this second step, on the rising edge of the next clock pulse, the ALU output will be stored in the Accumulator and the carry-out will be stored in a one-bit carry flip-flop. When the ADD operation is complete, the control logic will instruct the data path to begin another instruction fetch/program counter incrementation cycle, and execute the next instruction found in the computer memory.

We need to expand our drawing of the data path to show the other registers and pathways involved in the ADD operation. First, we will add the components needed for the Data Register fetch step.



Notice that I added the Address Source Multiplexor to select the source of the memory address for the operand fetch. During the instruction fetch, the Program Counter held the address of the instruction. Now, the Instruction Register holds the address of the data to be placed in the Data Register for addition. Of course, the multiplexor input is selected by a signal from the control logic. At the end of this data fetch cycle, a register write pulse is sent by the control logic to the Data Register, but no write pulse is sent to the Instruction Register. This ensures that the Instruction Register will continue to hold the ADD instruction that was fetched earlier.

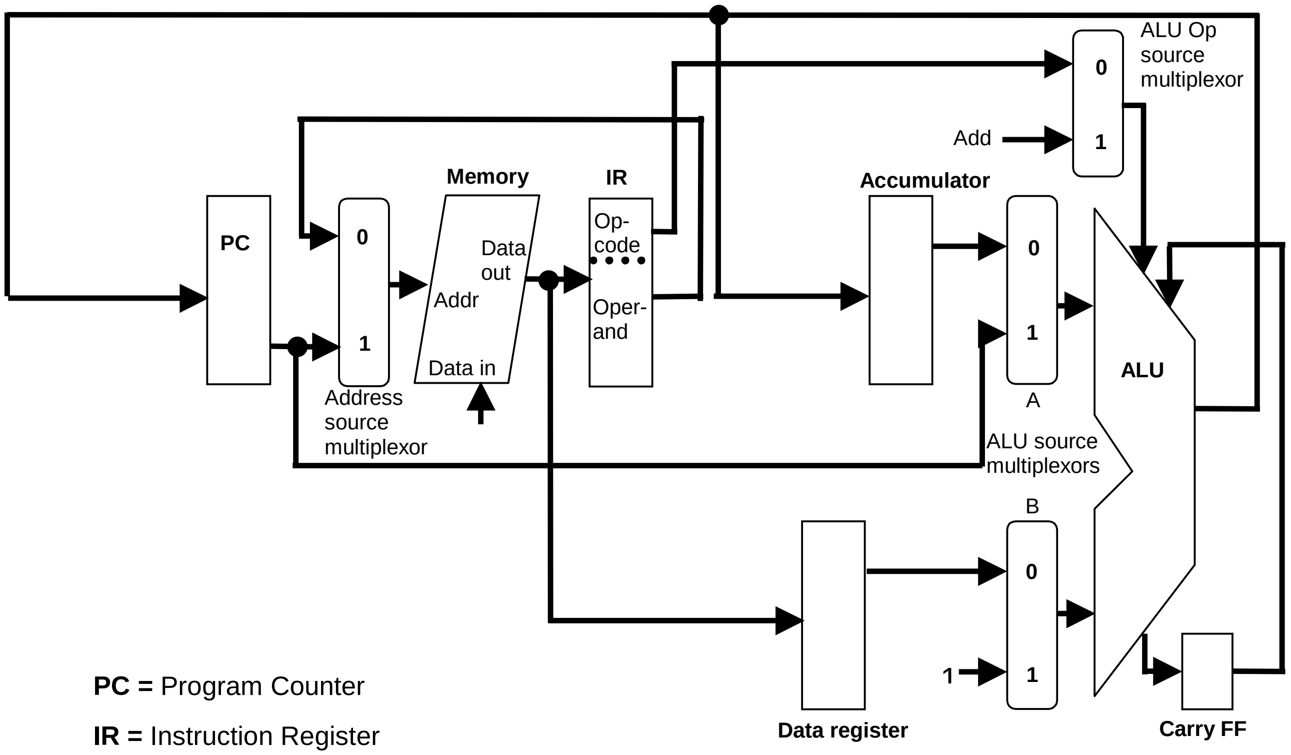
During the final cycle of the ADD instruction, the contents of the Accumulator and Data registers are directed to the ALU inputs, along with the ADD opcode contained in the IR bits 12-14. After a short delay, the result of the addition will appear on the ALU output. This result will be clocked into the accumulator register by a write pulse at the end of the cycle. The carry-out bit will also be stored in a flip-flop. We will now add the Accumulator register and a few more elements to the data path diagram.



We need ALU source multiplexors to send the proper inputs to the ALU. In the Program Counter Incrementation step we sent the Program counter to the ALU A input, and the 12-bit value 1 to the ALU B input. In all the two-input arithmetic and logical instructions, such as ADD and OR, we will send the Accumulator to the ALU A input, and the Data Register to the ALU B input. We also need to add a multiplexor to select the ALU operation, either the stand-alone ADD operation for the PC incrementation step, or the ALU opcode contained in bits 12 to 14 of the Instruction Register.

Notice how the ALU output is connected to both the Accumulator and Program Counter inputs. At the end of the last cycle of execution of the ADD instruction, the control logic sends a write pulse to the Accumulator and carry flip-flop. No write pulse is sent to the Program Counter, so it remains unchanged. After this final write pulse, the Accumulator will contain the result of the addition, and the Carry flip-flop will contain the carry-out bit.

The data path diagrammed above is adequate for the instruction fetch/program counter incrementation cycle, and for all the arithmetic-logical operations except those using a carry-in (or borrow). To finish the data path for the arithmetic instructions that need a carry input, we add a path from the output of the Carry flip-flop to the ALU carry-in. Whether the carry-in is used or not depends on the ALU opcode, so there is no control line for this outside the ALU.

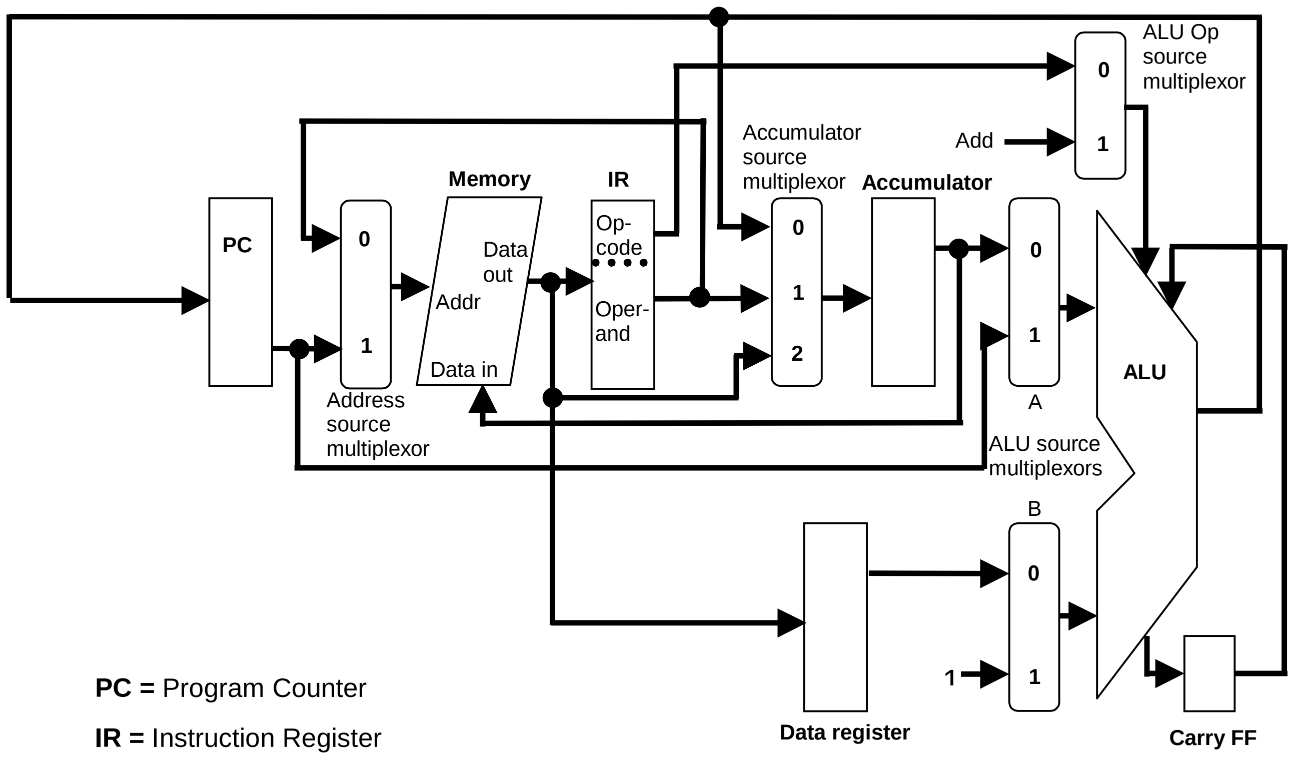


PC = Program Counter

IR = Instruction Register

We now have a data path that will perform all eight of the arithmetic-logic instructions in our instruction set. A few more modifications will allow the other eight instructions to be performed.

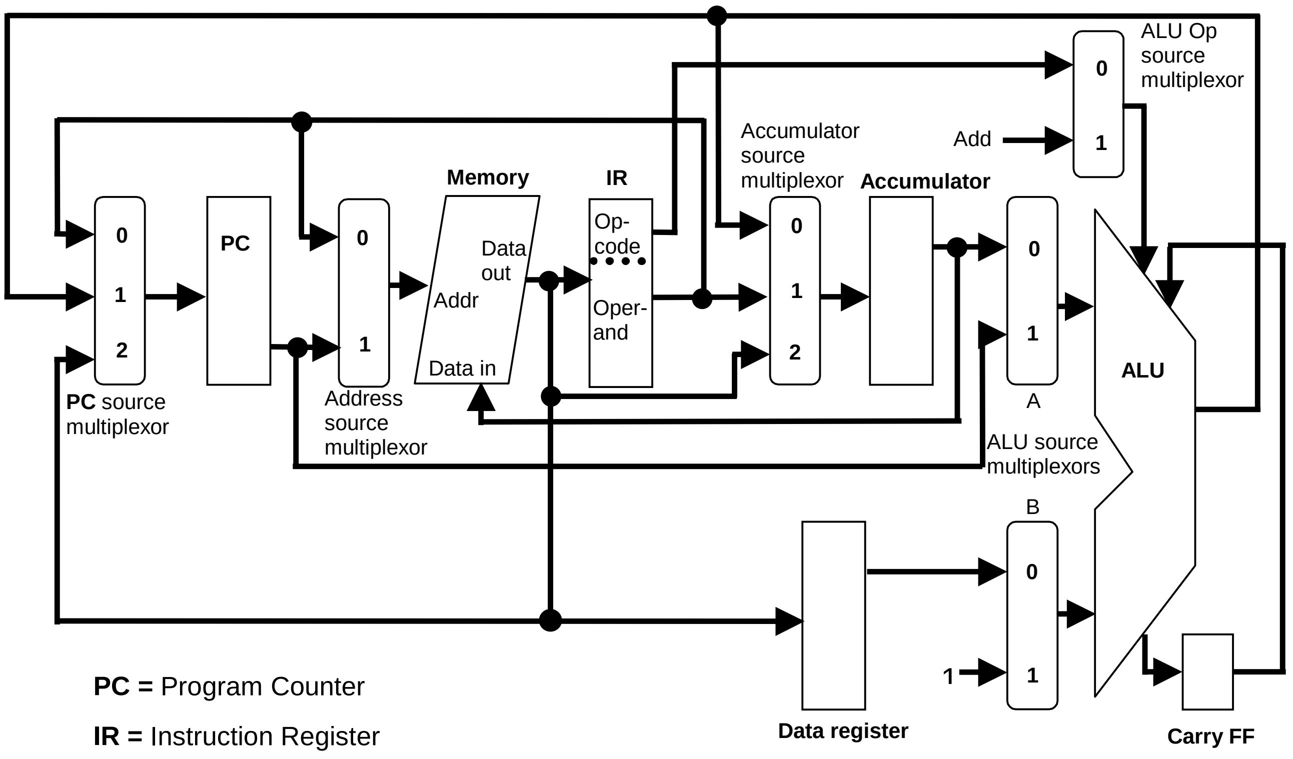
To perform the load/store instructions, connections between the accumulator and the memory are needed. There is only one type of store instruction, that is, store accumulator to memory (STM). However, there are two types of load instructions: load accumulator from memory (LDM), and load accumulator immediate (LDI). In the LDM (memory) load, the accumulator gets its input from the memory; the memory address is in the lower 12 bits of the instruction register. In the LDI (immediate) load, the value itself is in the instruction register. Therefore, we need an Accumulator Source Multiplexor to select the appropriate accumulator input: the ALU (from the arithmetic-logic instructions), the memory (LDM) or the instruction register (LDI).



PC = Program Counter

IR = Instruction Register

This data path is able to execute the instruction fetch/program counter step, the arithmetic logical instruction steps, and the memory load/store instruction steps. The remaining instructions are the program flow, or jump instructions. These instructions operate by placing into the program counter a new value, either from the memory (indirect jump) or from the instruction register (direct jumps). Therefore, we need connections between the instruction register and the program counter, and the memory and the program counter. This means we need a PC source multiplexor to select the proper PC input. The conditional jumps in our limited instruction set are direct jumps, and whether they are executed or not depends the value of the carry flip-flop (jump on carry) or the value of the accumulator (jump on zero, or jump on minus). No extra connections are needed beyond those for the unconditional jumps. If the condition is met, the control logic will cause the (direct) jump to occur, and if not it will simply cause the data path to fetch the next instruction in memory, skipping the jump step.



This completes the data path for the computer. There are some details that need to be added. Extra logic is needed in order to work with real computer memory that doesn't write like a register. This topic deserves its own section, which will follow. Also, some simple logic (11 two-input OR gates) is needed to derive the accumulator zero signal (the accumulator minus signal is simply the uppermost bit in the register). If you can understand the diagram above, you are well on your way to understanding a real computer at its most fundamental level.

This data path can be built with standard IC components of the 74LS00 series. Specifically, there is an IC that has an 8-bit register in a 20-pin package, and two of these will do for the instruction register. Similarly, three 6-bit register IC's will make a 12-bit register. The 12-bit 4 input multiplexors can each be built of 6 dual one-bit multiplexor IC's, and the one-of-two multiplexors can be built from IC's that have four single-bit one-of-four multiplexors per chip. To make the zero logic, a 12-input OR gate can be used, made from 11 2-input OR gates on 3 IC's. One chip with the carry flip-flop is also needed. The total number of IC's needed for this data path is 34, and would cost about $17.00. Of course, the sockets and board increase the cost up to about $50.00.

Real Computer Memory

As mentioned in the previous section, real computer memory does not operate like a large collection of data registers. There are two main differences. First, the memory write process is more complicated, involving several signals that have to be created by the computer logic. Second, memory uses a bi-directional data path, called a data bus.

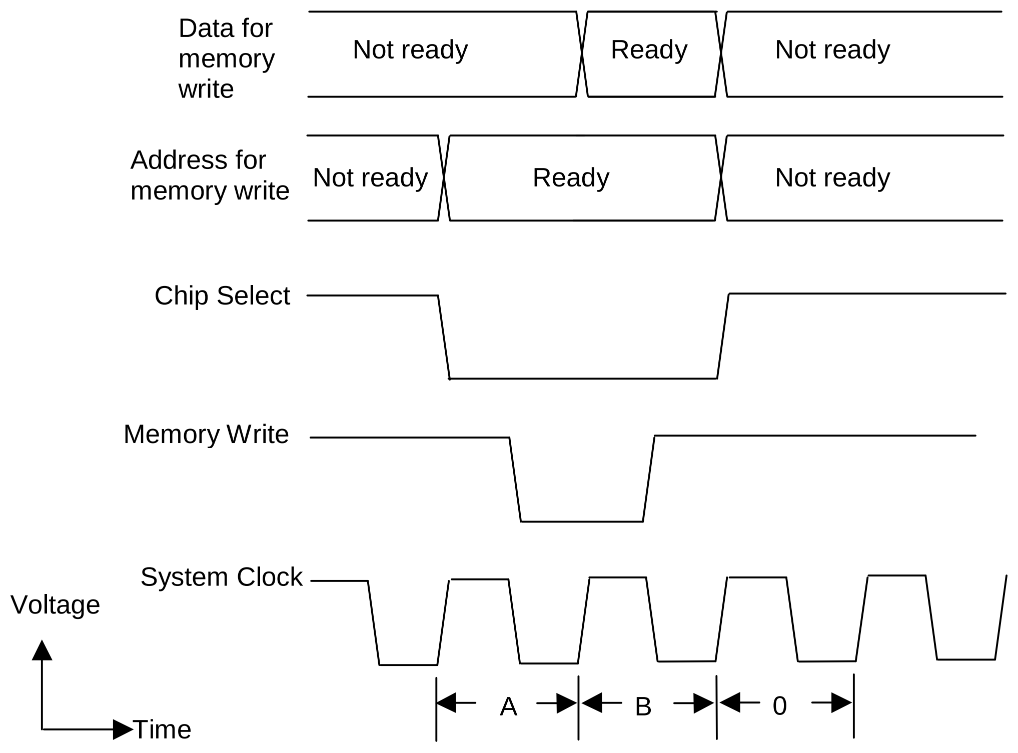
The semiconductor RAM memory circuit I used in this computer is 4-bit by 1K static random access memory (SRAM), specifically the 2114 IC. Static RAM does not require refreshing every millisecond or so like the cheaper, faster dynamic random access memory (DRAM). It is important to build your hobbyist computer with SRAM, because in the troubleshooting phase of construction you will need to run it very slowly, one cycle at a time, in order to find out why it is not working. DRAM will not function in a single step cycle, unless it has its own clock circuit. An additional benefit of using SRAM is that you don't have to create the circuitry to perform the refresh cycles during normal operations. Now, one can buy DRAM with the refresh circuitry built in, called SDRAM. However, our computer uses so little RAM, that it makes sense just to use the old-fashioned 1K 2114 circuit. It is all the RAM you will ever need.

The memory is written with the help of two inputs. The first is the chip select (CS) input, which must be made low (0 V) in order for the chip to function, either for input or output. The other input is the write input (WR). This input must be made low, and held low for a specified minimum time. During this time, the address and data on the memory inputs must be held steady. At the end of the required time span, the write input is made high (5V), and the data is locked into the memory.

The key factor in the timing is that the memory write input must go high before the address or data inputs change, in order to ensure that the data is safely written. This presents a problem for the computer designer. He (or she) would like to write to the memory in a single cycle. However, at the end such a cycle, the address, data and memory write signals would all change at approximately the same time, putting the data in some peril. You might try to chance it, but there is a better way.

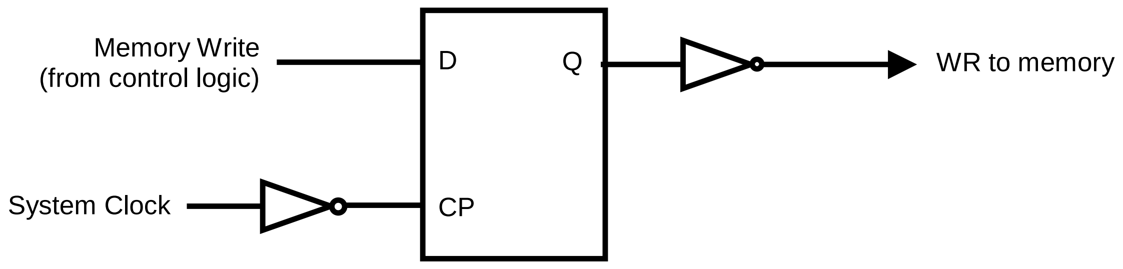
The answer is to spread the memory write over two clock cycles. This gives us some additional clock edges with which to work. We can now ensure that the memory write goes high well before the data or address inputs have a chance to change.

Here is the kind of timing we would like for a solid memory write:



Each figure above represents the voltage level on the indicated memory inputs, or the clock. The memory write process proceeds this way. At the start of the first memory write clock cycle, A in the diagram, the processor puts the address of the memory location to be written on the address inputs of the memory. Simultaneously, the chip select memory input is made active (low) since this memory control input is derived directly from the address. One-half cycle later, on the downward edge of the clock in cycle A, the Memory Write (WR) is made active (low). At the beginning of the B cycle, the data is placed on the memory inputs, and on the downward edge of the clock in the B cycle, the WR input becomes inactive (high). At this point, the data is locked into the memory. At the end of the B cycle, on the upward clock edge of the next machine cycle, the address used during the memory write, and the derived CS input, are inactivated. The address, CS, data and WR are all active during the first half of the B cycle, which lasts 500 nanoseconds for a 1 MHz clock. This easily meets the minimum time requirement for writing the 2114 memory chip, which is about 200 ns.

It is not hard to make a circuit that will provide the proper timing of the memory write input (WR). Here it is:

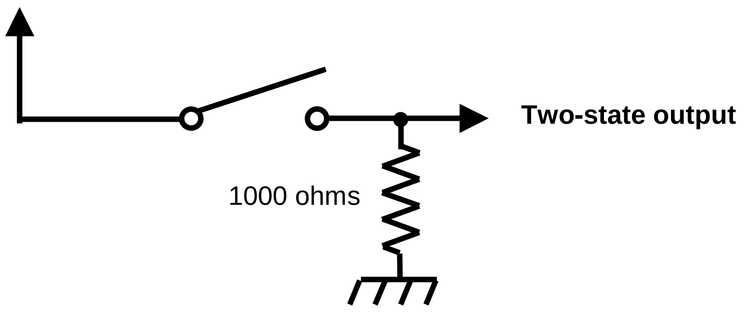


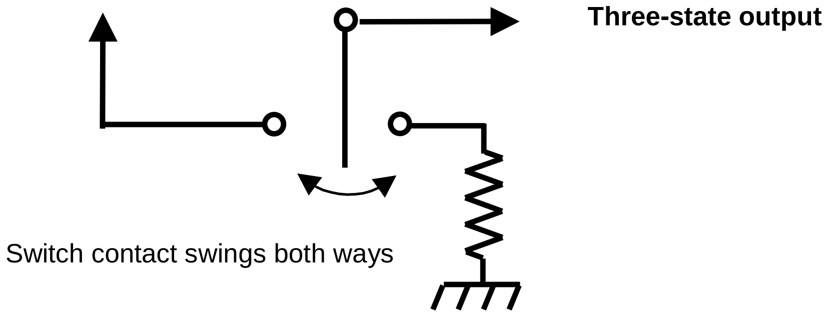
This is simply an edge-triggered flip-flop whose clock pulse input is the inverted system clock. The control logic Memory Write signal comes on (becomes one) at the beginning of the A machine cycle, in response to the program instruction STM. However, since the flip-flop has an inverted clock signal, it is not written into the flip-flop until the middle of the A cycle. That is, when the clock has a falling edge in the A cycle, the inverted clock input to the flip-flop will have a rising edge, and the Memory Write signal will be written into the flip-flop. It then appears on the output, and is inverted. This output is connected to the WR input on the memory chip (the WR signal to the memory chip is active when low.) When the WR memory input goes low, the memory begins to store the data that the processor has placed on its inputs. The Memory Write signal from the control logic goes to zero at the end of the machine cycle A, but since the flip-flop has an inverted clock input, the WR output is held low for another half-cycle, giving the memory the time it needs to finish storing the data. In the middle of the B cycle, the flip-flop sees a rising clock edge from the inverted clock input, and stores the Memory Write zero. The WR now goes to 1, and the memory write is finished.

The timing diagram shows the requirements for writing a real semiconductor memory with the various control inputs. The other feature of memory, that makes it different from a register, is that it has bi-directional data input/output lines. These lines are controlled by a device called a three-state buffer, which I will now describe.

Recall that a computer consists of large networks of automatic switches, each of which is either on (logical 1, or 5V) or off (logical 0, or 0V). But remember, that in order to be either 5 or 0 volts, an output has to be connected, through the logic gate circuit, to either the 5V or 0V (ground) power supply leads of the gate. What would be the state of an output that was connected to neither? That is, pretend you cut the output wire, and it is hanging in the air.

Remember that the output voltage of a logic circuit is only part of an output. The other is the ability to pass current. So, if an output is 5V, it also needs to be able to pass some current to drive inputs of other gates to which it is connected. Similarly, if an output is 0V, it needs to be able to "sink" current, in order to operate the circuits it is tied to. The "cut wire" state is a third state, neither 1 nor 0, that is very useful in computer system design. The third state is also called "high impedance", because current will not be able to flow either into or out of an output that is in this state. The high impedance state can also be thought of as having a very high resistance. It is shortened to "Hi Z". Here is a simple switch diagram that shows a two-state and a three-state device.



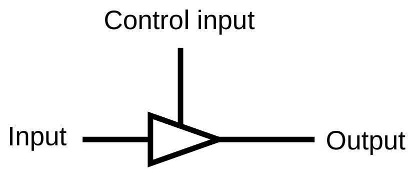


In the upper diagram, the switch can be open or closed. This type of switch is called a single-throw switch. If open, the output is connected through the resistor to ground, and will have a voltage of 0 with ability to sink some incoming current to ground through the resistor. When the switch is closed, the output will be 5V, with the ability to pass significant current to outside devices. The lower diagram shows a switch that can have three positions; closed to the 5V contact, open in the middle, or closed to the grounded contact. Such a switch is called a double-throw switch. Note that the circuit output is connected to the central pole of the switch. The middle position is the third state. It is just like a cut wire. The output, connected to the central pole of the switch, will not be able to conduct any current anywhere with the switch in the middle position and its voltage will be undefined. This is characteristic of the third, or high-impedance state.

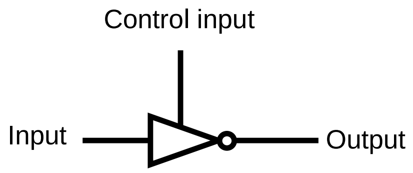
Having a three-state logic device output allows us to take a huge short cut when we assemble a true computer system. It allows us to use a collection of wires to pass data in two different directions. A collection of parallel wires is called a bus, and when it can pass data in two directions it is called a bi-directional bus.

The bi-directional bus allows us to connect many separate devices that have three-state outputs to the same set of wires, and then use logic to select which devices will communicate to each other. The other unselected devices will remain in the third state, and will be invisible to the active circuits. They will not interfere with the data communications between the active devices. We are bringing this up here, because computer memory is such a device, with three-state data outputs. When the memory write signal is given to a selected memory chip, the outputs behave as data inputs, in order to write data in the memory. When the memory write signal is inactive, the outputs behave as outputs, sending data onto the bus. When the chip is not selected, the outputs are in the third, high-impedance state, and the chip is invisible to other devices on the bus, such as input or output ports. It is convenient to think of the memory data lines as input/output lines, because they can change direction.

It is a simple matter to make a bi-directional bus using three-state logical devices. The two most commonly used are the three-state buffer, and three-state inverting buffer.

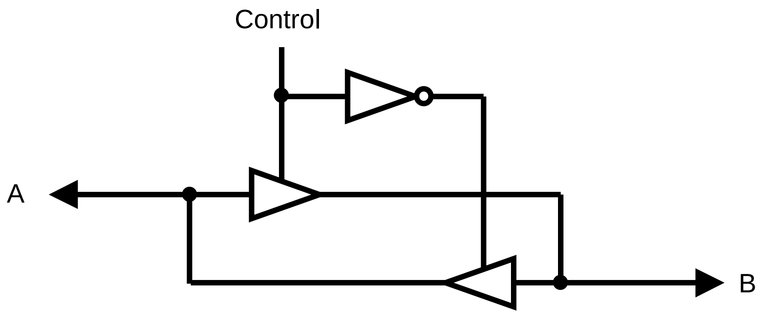


In	Control	Out
0	0	Hi Z
1	0	Hi Z
0	1	0
1	1	1



In	Control	Out
0	0	Hi Z
1	0	Hi Z
0	1	1
1	1	0

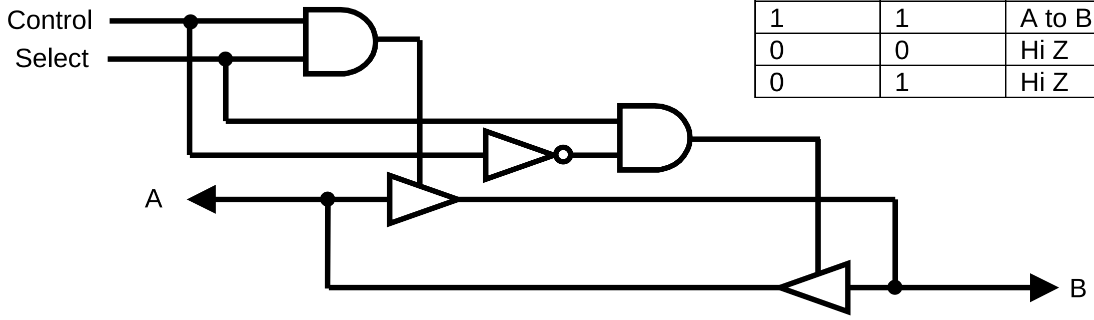
The control input is a signal that enables the gate to pass data when it is on (1). The control input is sometimes called an enable input. Here is how to make a bi-directional buffer out of two three-state buffers and an inverter.



When control is 1, the data flows from A to B, and when it is 0, from B to A. In other words, when control is 1, A is the input and B is the output, and when control is 0, B is the input and A is the output. Thus A and B are input/outputs.

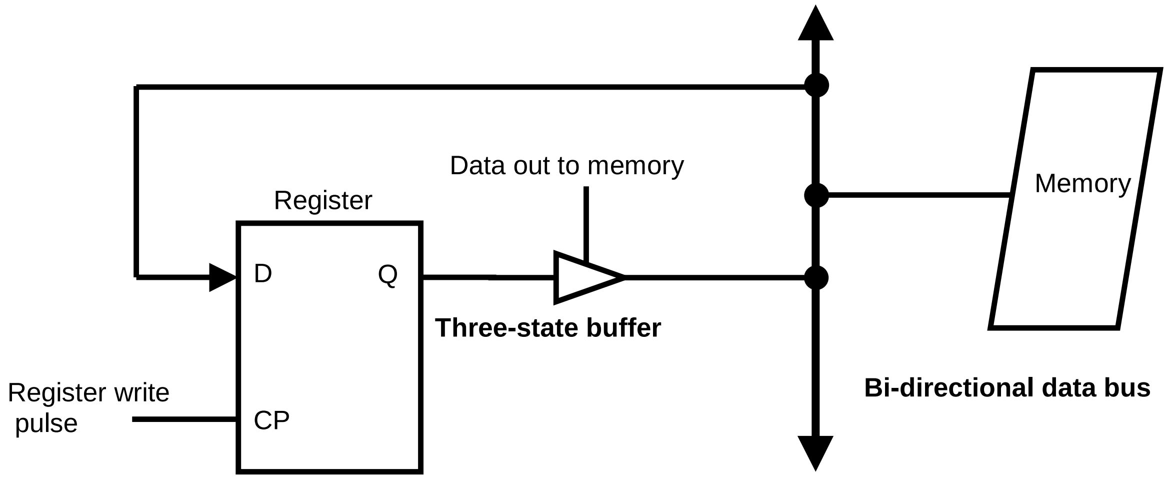
Inclusion of another control can allow the bi-directional buffer to go to the Hi Z state.

Select	Control	Direction
1	0	B to A
1	1	A to B
0	0	Hi Z
0	1	Hi Z



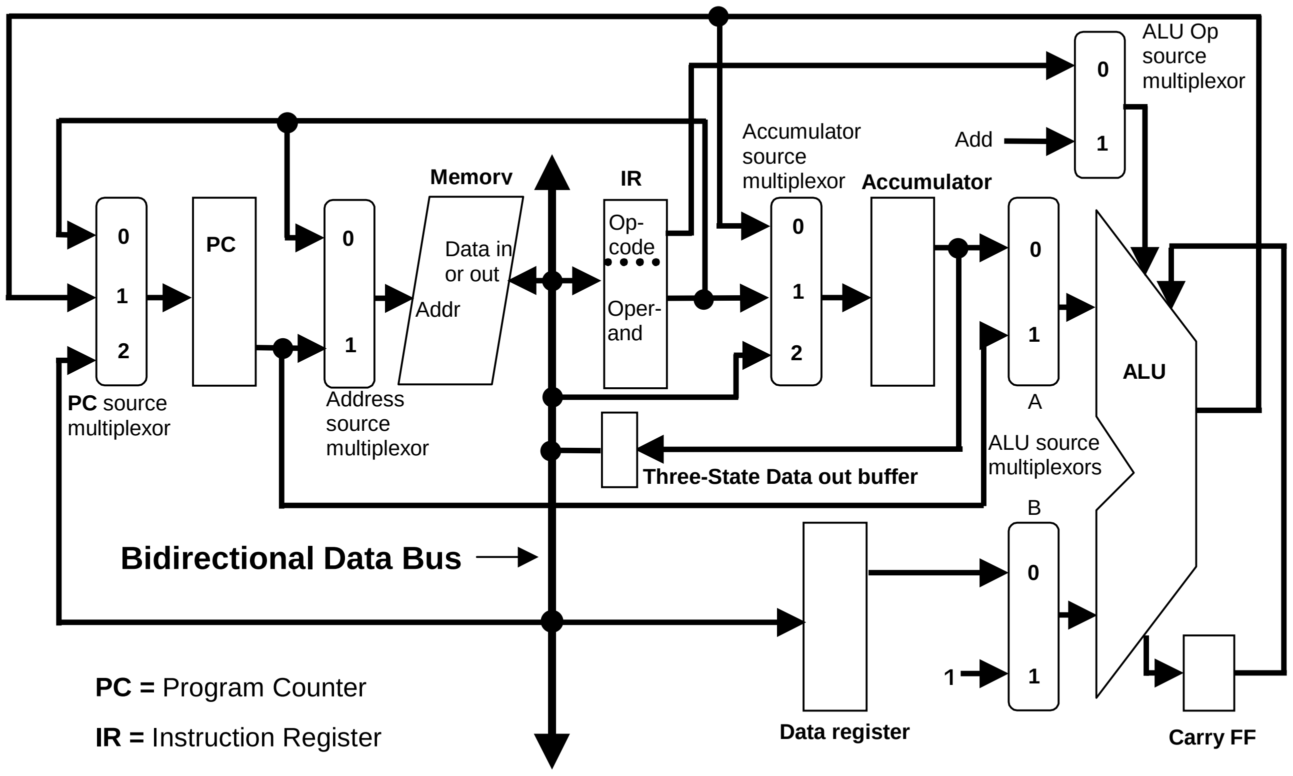
This type of circuit is used inside memory circuits to control the input/output lines

In the data path defined in the previous section, we are faced with the problem of how to connect a register, with separate inputs and outputs, to a memory chip with bi-directional three-state input/outputs. First of all, the problem really only occurs during the memory write cycle, because otherwise the memory is either putting out data (if selected) or is in the high impedance state (if not selected). If the memory input/outputs are directly connected to register inputs, both of these conditions are tolerable. However, the register outputs cannot be connected directly to the memory input/outputs, because when the memory is in the output condition, you will have memory outputs connected to register outputs, which is not tolerable. We need to use a three-state buffer to prevent the register outputs from colliding with the memory outputs. The circuit is simple:



The three-state buffer protects the memory outputs from colliding with the register outputs when the memory is in the output mode. When the memory is in the input mode, the Data out control signal comes on, and the three-state buffer becomes an output. In the timing diagram shown earlier, this takes place at the beginning of the B machine cycle.

Here is a diagram of the data path with the three-state data out buffer included. Now, the memory input/outputs are connected to a bi-directional data bus.



The only devices capable of putting an output on the data bus are the accumulator (during a memory write) and the memory (most of the rest of the time). The other devices (multiplexors and registers) are always inputs. Inputs don't interfere with bus traffic.

We can put other devices on the bus, such as input/output ports. Any device that places data onto the bus, like an input port when active, will need to have three-state outputs and proper logic design to control them. As long as the devices that place data onto the bus have three-state outputs, there is no limit to the number of devices that can use the bus.