7: Microarchitecture

7.1: Where are we now?

We started with two organizing questions: "How do we program a computer?" and "How do we build a computer?" In chapters 2 and 3, we learned to program a computer using Python, and in chapters 5, we learned how to translate our Python programs into the computer's native language. In chapter 6, we learned that this native language, while we can write it using mnemonics for improved human readability (well, some improvement in readability anyway), is actually just numbers.

So now that we know the computer's native language--that is, all commands are 16-bit numbers, and we know which of those numbers correspond to what operations, our job in this chapter will be to start down the path that answers the second question: "How do we build a computer?" But at this point, we can actually understand the task more precisely:

"*How do we build a physical machine that can take physical representations of numbers that represent commands in the ISA and perform the corresponding operations of moving physical representations of numbers between various physical storage units?*"

The first step to answering this is called the computer's **microarchitecture**, which provides a high-level view of the machine we will build in terms of high-level components--think at the level of "bakery" and "delivery squad" back in our Waffle company. In later chapters we'll get into the details of how those components themselves are put together (more like the level of oven temperatures and what color hats the chefs should wear).

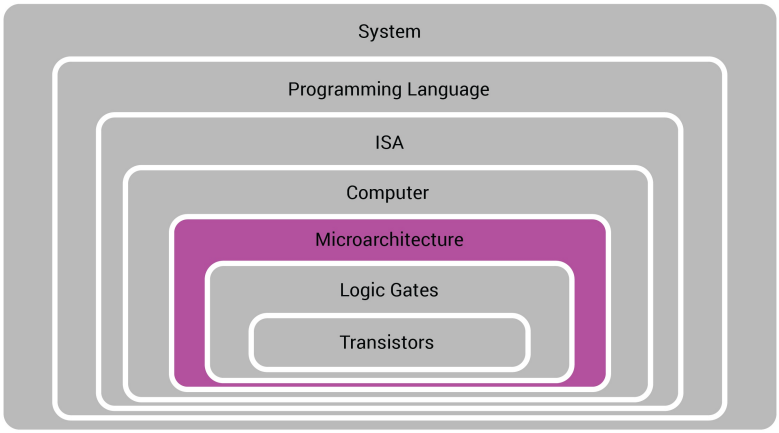

Figure 7.1.1:

7.2: What is a microarchitecture?

Summary:

Definitions: microarchitecture

By now we've understood that to build complex systems, all we need is a machine that can execute commands from the ISA. Further, we gave the ISA commands binary representations so that all our machine has to do is to accept these as its inputs and perform the corresponding operations. That is, the ISA--with its instruction set and encoding--was the computer's interface. A **microarchitecture** is an implementation of that interface.

As with most of the implementations we've discussed before, the the implementation of a computer will be done in layers: We will discuss it in terms of high-level components in this chapter, and then work our way down to the wires and switches and electrons in the next chapter.

In this chapter, our implementation will happen in two steps:

1. First, we will describe a computer as a specific kind of abstract machine called a **state machine**.
2. Then, we will start to build this state machine as a physical device using various **circuit components**.

That is, we will first work toward constructing a large state machine that completely describes the behavior of our computer. Then we will turn this into a circuit diagram for our computer consisting only of various circuit components and wires connecting them. This will mean that to actually build the computer, all we will need is to build the individual circuit components, which will be the topic of the next chapters. At that point, upon hooking them up as in the diagram, we will have designed our own computer!

7.2.1: State machines

As we said: our first step will be to implement our computer as a state machine. A state machine, broadly, is a description of a system as a collection of states it can be in, and how it changes to different states depending on input. Specifically, it encompasses the following three pieces of information:

1. The states: What possible states are there? These should be exclusive (it cannot be in two states

at once) and exhaustive (it must always be in one of these states).
2. The inputs: What possible actions can be performed in the system?
3. The transitions: How does each action affect the state the system is in?

Let us take, for example, a simple state machine modeling the behavior of a door. We need to provide the three pieces of information described above:

1. States: It has three possible states: Open, closed, locked.
2. Inputs: At any time, the actions available are: Open, close, lock, and unlock.
3. Transitions: How the actions affect the state

In our example, to describe the transitions, we now have to enumerate the possibilities: If the door is open, then opening the door does nothing new and so leaves it in the state of being open, closing the door moves it to a state of being closed, and locking and unlocking the door doesn't work since it is open, and so the door stays in the state of being open. Then we'll have to describe also what the four actions do to a closed door and to a locked door, but this will be rather tiresome. So we come to a much more efficient and readable way of presenting this same information, which is the picture referred to the state diagram for the state machine. In it, we draw each of the possible states inside circles:

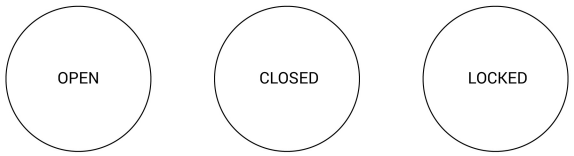

Figure 7.2.1.1:

And then, to indicate the actions and their effects on the states, each state gets one arrow going out of it for each possible action. This arrow will be labeled with the corresponding action, and will terminate at the state that the action leaves the system in. In the case of the door, we get this diagram:

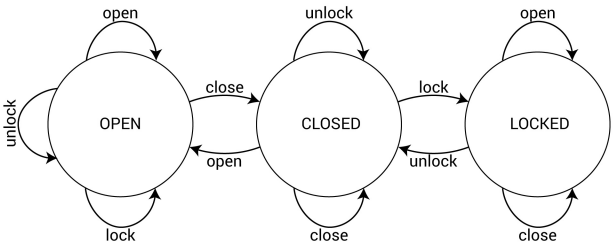

Figure 7.2.1.2:

So if the door is closed and we perform the "lock" action, then we find the arrow labeled "lock" starting at the "closed" state. This arrow terminates at the "locked" state, telling us that we end up in the "locked" state.

As a convenience, sometimes we omit the arrows that start and end in the same state, drawing the previous diagram instead as:

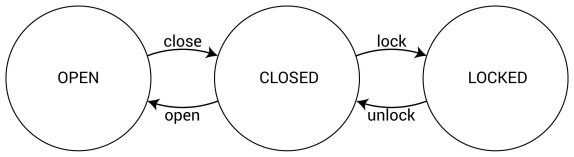

Figure 7.2.1.3:

where the fact that, e.g. if the door is open, then locking doesn't change its state being left implicit.

Of course, some doors do not behave this way. Instead, they have a deadbolt that deploys when the door is locked, meaning the door cannot be closed. To describe this, we would use states: "open with deadbolt retracted", "open with deadbolt deployed", "closed, deadbolt retracted", and "closed, deadbolt deployed":

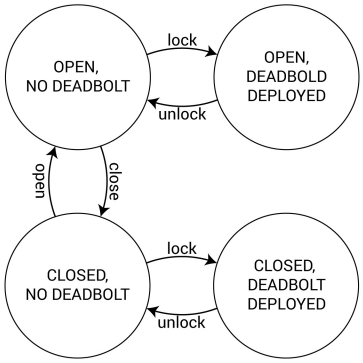

Figure 7.2.1.4:

Other doors have a lock that affects only whether you can turn of the knob, meaning they can be locked while open, and if they are subsequently closed, they will be closed and locked. These have a slightly different state machine, wherein you can close the door from its "open but locked" state:

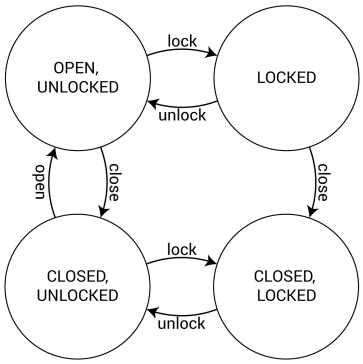

Figure 7.2.1.5:

7.2.1.1: A state machine for our microarchitecture

To describe our microarchitecture as a state machine, we need to describe the states, the input, and the transitions. The inputs will be all the possible instructions, and each state will perform a single small operation, e.g. "write this value to that memory" or "read this value from one memory and store it in some other memory" or "add these two values".

As we execute any given instruction, we will go through several states. For instance, ldi will first go through a state that figures out what value you want to store, then another that figures out what register you want to store it in, and then a state that actually stores that value in that register. Finally, it will need to go to a state that will get the next instruction.

The states each instruction passes through will be different, but there will be a common pattern of five stages of execution that the states will fall under, called respectively:

1. The "fetch" stage: These are the states that retrieve the next instruction we are supposed to execute.
2. The "decode" stage: These states will figure out what the instruction does--e.g. for ldi, which value it wants to store and which register it wants to store it in.
3. The "execute" stage: These states will perform any arithmetic requested by the operation. For example, ldi will not have any states in this stage.
4. The "memory" stage: These states will perform any RAM-related operations the the instruction requires. ldi will not have any states in this stage either.
5. The "writeback" stage: These states will perform any updates to registers. ldi will go through a state here that writes the value to the desired register.

In more detail:

Fetch: Remember that the value stored in the PC is the address of the current instruction we are

meant to be executing. So in the fetch stage, we feed the value from PC into the program memory to read the actual instruction from that address.

Decode: Having got the current instruction--that is, the 16 bits encoding it, we need to figure out which instruction it is, and then what its various bits mean. For instance, if we get 1110010100011001, we can decode this as an ldi instruction that should write value 01011001 to register 10001. We don't at this point actually store anything in register 10001, but merely store the information of what value needs to be written and which register needs to be used.

Execute: Now that we've separated out the information relevant to the instruction, in the execute stage we perform any computations required by the instructions. For instance, if the instruction was add r20,r30, the decode stage will have determined that we're storing to r20, and will have read out the values from r20 and r30 that we're going to add. In the execute stage, then, we actually add the two values. This will also involve computing the new SREG--e.g. if the result of the addition is 0, we will set the Z flag.

Memory: Some instructions, notably ld and st, involve a trip to RAM. In this stage we perform any operations related to RAM. So if we're running ld r30,567, then decode will have determined that we're reading from address 567 and storing into r30. The memory stage, then, will actually read out the value from data memory at address 567.

Writeback: In the final stage, we perform any modifications to internal memory slots--that is, to the register file, PC, and SREG. So, for instance, when running add r20,r30, decode will have determined what values to add as well as that we're writing to r20. Execute will have actually added the two values, so now we have the value we need to store (the sum) and where to store it (r20), so in this stage we actually store the sum to r20, and the status flags computed in the execute stage will be stored into SREG.

One thing that we left implicit in our descriptions above is how data gets passed between the stages. For instance, if the instruction is ldi r17,89, then the decode stage, will determine that register 17 needs to be written to and the value 89 needs to be written. These numbers then need to be available to the writeback stage, which will actually store the value 89 into register 17.

For this purpose, the microarchitecture will introduce some **auxiliary registers**--that is, named memory slots where any given stage can store values and a later stage can read those values out. For example, we will have one called REG and one called VAL. When decode gets ldi r17,89, it will store the number 17 into REG and 89 into VAL. Then the writeback stage for ldi can be simply implemented telling the register file to write whatever is stored in VAL to the register at the address stored in REG.

7.2.2: Circuits

The method we will use for physically constructing our computer will be electronic circuits. At a high-level, circuits are made up of **components** connected with **wires**. Components are black boxes of circuitry that we specify in terms of their interface only: That is, they have some specified numbers of input and output wires, and we specify what the outputs do when certain inputs are provided.

For us, each input and output will be given a name and will consist of a collection of wires--i.e. channels that can carry electrical current. The number of wires in a particular input or output is called its **width**. So, for example, we can imagine a component with one input called IN of width 3 and one output called OUT of width 1. We could draw this like so:

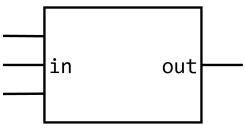

Figure 7.2.2.1:

But sometimes drawing each of the wires involved in a given input gets tedious, so we may also at times draw the input of width 3 as a single wire with some notation indicating that that one line indicates three wires:

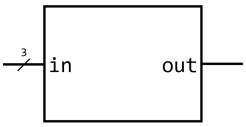

Figure 7.2.2.2:

Now, in general, wires can carry varying levels of electrical current, and components whose behavior depends on the precise amount of current on its input wires is called an **analog** circuit. Such a circuit may also send current on its output wires at varying levels depending on the input currents. All the components we will discuss here will instead be **digital**--that is, the output will only depend on whether each input wire has current flowing and will not care about how much, and the output wires will only either have current flowing on them or not--precisely how much current is neither specified nor important.

Warning:

Obviously this is not literally true--if the amount of current is small enough, then most devices will not be able to detect it and will treat it as though there is no current. On the other side, if the current is enormous it can overwhelm the device and cause unpredictable behavior or catastrophic failure. But most digital devices operate within some reasonable ranges in which these concerns do not apply. TODO: There's something about current vs voltage that might make this at least partly wrong; sort it out.

Since all of our circuits are digital, we can describe the state of the input wires at any time by saying which ones have current on them and which do not. There are a few short-hands for doing this: If an input wire has current coming in, we say that input is **high**, and if not, we say it is **low**. In a diagram, we might denote a wire that is high with a 1, and one that is low with a 0. For example, we can describe several possible situations the above example component might be in this way:

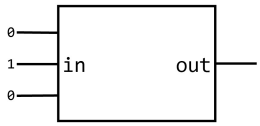

Figure 7.2.2.3:

With this, we can be even more brief in our description by treating each input's wires as the binary representation of a number. So if we instead represent each input and output with a single line in our diagram with a notation indicating its width, then we can represent the situation in the above figure by:

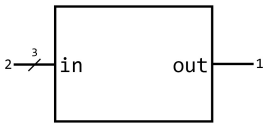

Figure 7.2.2.4:

We're writing the input value as 2, which in binary is 10. Since the input has width 3, we can read off what each individual input wire is doing by looking at the 3-bit representation of this, i.e. 010. So the middle wire is high and the other two are low. Likewise, the output wire is high.

Now, in our microarchitecture, we will make use of four primary types of components:

Memories: These will be circuit components with the following inputs: Address, Input data, and write enable. The address and data inputs will be the appropriate width for the size of the memory. The write enable input is one wire. When this input is high, the input data value will be stored in the slot given by the address input. When the write enable input is low, the data input affects nothing, and the address input affects the output. There is one output--the data output--which will always reflect the value stored in the memory slot whose address is given on the address input.

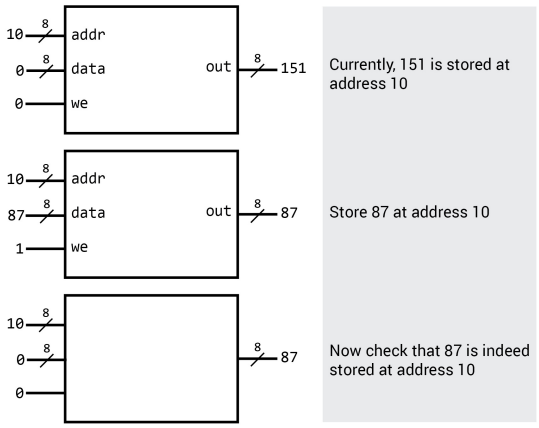

Figure 7.2.2.5:

Arithmetic logic unit: Remember how one of the basic things a computer can do is arithmetic? The **Arithmetic logic unit**, or **ALU** is the piece of circuitry that accomplishes this. It has four inputs: two called A and B respectively for the incoming numbers to be operated on, one called op to indicate what operation, and one called carry_in which is a slight extra thing so we can tell the ALU whether the previous operation resulted in a carry. This is important for some instructions, like ADC, but isn't important to think about most of the time.

Now recall that inputs to any circuit are just collections of wires, and in this chapter we're thinking about the values coming in on those wires as the binary representation of a number. In particular, the op input has to be a number. So we need to assign numbers to the various operations when designing the ALU. We can just make one up, but here is one that will turn out to be well-adapted to the ISA encoding (TODO: Check this):

Number	Operation
0	Addition
1	Addition with carry
2	Subtraction
3	Subtraction with carry
4	Binary and
5	Binary or
6	Binary xor
7	Binary nor

So for example, the component looks like:

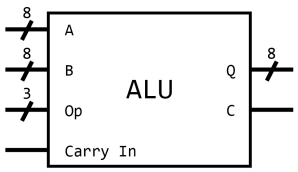

Figure 7.2.2.6:

And we can use it for the various operations by doing thing like:

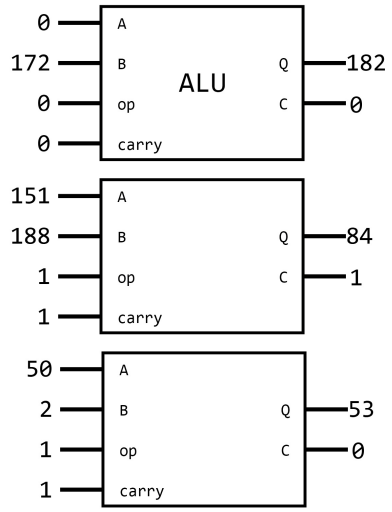

Figure 7.2.2.7:

State machine controller: As we said, a state machine is an abstract machine. So once we have our description of our processor as a state machine, we'll need to be able to convert this to actual circuitry. The circuit that does this is called a state machine controller. It has one input and some possibly large number of outputs depending on the state machine it is implementing. The way in which the outputs correspond to the states is complicated, and will be addressed when we describe our microarchitecture as a circuit later in this chapter.

Multiplexer: The multiplexer is perhaps the most basic piece of circuitry we'll describe, but also one of the most versatile. It is how you `make choices' in circuits such as `if the instruction is an add instruction I want to read some registers and put them through the ALU, but if the instruction is a st instruction then I want to write some register value to RAM instead'.

For such a daunting task, the actual specification of a multiplexer is quite simple: It has some number of inputs designated as data inputs of whatever width you like, one input called the `selector', and one output, whose width is the same as that of the data inputs. A multiplexer with N data inputs all of width W would be called an `N-by-W multiplexer'. The output's value is found by taking whatever's on the selector input--say this is x--and taking the xth data input as the output.

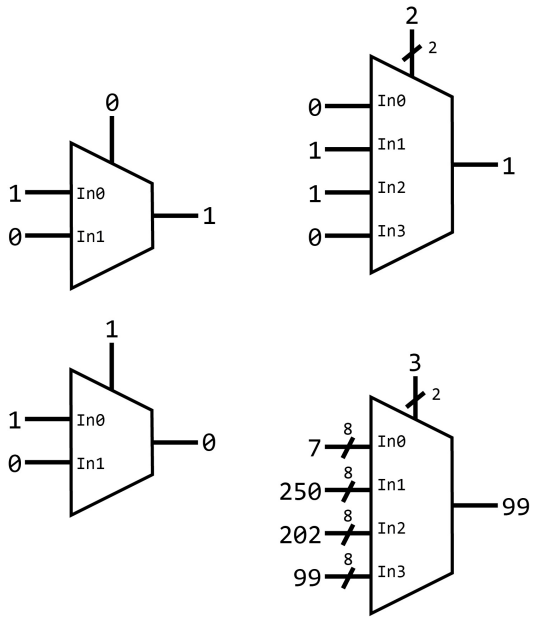

Figure 7.2.2.8:

Of course, using this device for the decision-making in circuitry is a bit of a trick, and we'll discuss that in detail later, but you should already get the idea that this might be the circuit for the task, at least.

7.3: Our microarchitecture as a state machine

Earlier, we described at a high level the sequence of operations that go into executing an instruction: We fetch the instruction from program memory, decode what precisely it means, perform any computation and memory operations it requires, and then save any results where needed.

Now we can flesh these out into precise sub-steps using a state diagram as the main organizational tool. The input to the state machine will be the current instruction, and the states will be various small operations, organized into the five stages outlined above, which when performed in sequence accomplish the job of the input instruction. Thus, when the input instruction is ldi, this will cause the state machine to go through a particular sequence of states--i.e. of operations--that will cause the correct thing to happen (an immediate value being loaded into a specified register), and then, once this is done, the final state will cause the input to the state machine to be changed to the next instruction, whereupon the process will begin anew.

In order to start constructing this state machine, then, we need to know first: 'What sorts of operations is a state allowed to do?'

The answer, for the moment, is very simply: Read and write to any of the available memories, and perform computations on any values we get. So we can include the operation 'Write the number 2 to REG', or 'Write the number 67 to r11' or 'Add 2 and 5 and store the result in memory at address 463'. However we can also take values from other memory slots and use those as addresses or values for memory operations. For example, 'Write the number stored in VAL to register with address REG' is also a valid operation. We will abbreviate this as RF[REG] = VAL. We can also do things like 'Add VAL and VAL2 and store the result in VAL'. (The rules for exactly what operations are allowed are quite vague because really the rules are 'anything for which we are willing to create a piece of circuitry'. But if we maintain a reasonably simple set of operations, then our circuitry will be relatively simple.) So a valid set of operations to perform in a single state is: VAL = VAL1 + VAL2, and REG2 = RF[REG1].

The sort of things we want to avoid having together in a single state include multiple operations where the input of one depends on the result of another. So, for instance, REG = VAL1 + VAL2 and VAL = RF[REG] could not go in the same state. The reason for this will become more clear when we discuss the workings of memory circuits in detail in the next chapter, but in brief, the idea is that all states are circuits, and there is a prescribed amount of time set in which the electrons should all traverse the circuit, and the correct answer (VAL1 + VAL2 and RF[REG1], in this case) come out as the outputs of the circuit. It is only right at the end of this prescribed time slot when the auxiliary registers REG and VAL actually get written, and they get written simultaneously. Thus the output of the RF[REG] circuit will be the value in the register file at register number 'whatever was in REG before this state started running', rather than at register number VAL1 + VAL2, because that sum will only be written to REG at the end of the circuit's time slot.

7.3.1: ldi

Let us start by constructing a state diagram for a machine that only understands ldi instructions. First, we need to decide on the sequence of operations that need to happen in each stage.

Fetch: The first stage that runs is fetch. This is meant to retrieve the current instruction from the program memory. Recall that the address of the current instruction is stored in PC, so it will read program memory at address PC. However, future stages will obviously need access to the instruction that we get back from program memory, so we need to store it somewhere. We will have an auxiliary register, then, for this purpose, called INST. Thus the only operation in fetch is:

```
INST = PM[PC]
```

Decode: Having got the instruction, which we are supposing for the moment is an ldi, we need to deduce from it what value will need to be written to which register. This is certainly possible by looking at the value of INST, so we read from INST the register to store to and the value to store and we store these in REG and VAL, respectively. Thus:

```
REG = the bits of INST that say which register to write to
```

followed by:

```
VAL = the bits of INST that say which value to write
```

As a bit of notation, we know from the previous chapter which bits these are, namely REG = a 1 followed by bits 7, 6, 5, 4 of INST. We will denote this like

```
1,INST[7:4]
```

Further, VAL is bits 11-8 followed by bits 3-0 of INST, which we will denote like:

```
INST[11:8],INST[3:0]
```

So in this shorthand, the two operations we perform, in order, are:

```
REG = 1,INST[7:4]
```

and then

```
VAL = INST[11:8],INST[3:0]
```

We'll say in the next section how one actually goes about doing these with real circuitry, but for the moment let us leave it like this and move on.

Execute: No arithmetic actually happens in an ldi instruction, so this stage does nothing

Memory: We do not touch the data memory in this instruction, so this stage also does nothing

Writeback: Finally, this is the stage where we are supposed to write any values to the register file, SREG, and PC. Of these, SREG does not need updating, PC needs to be incremented, and the register file needs to be written to. Specifically, the register at address REG needs to get value VAL. Thus:

```
RF[REG] = VAL
```

and then:

```
PC = PC + 1
```

In summary:

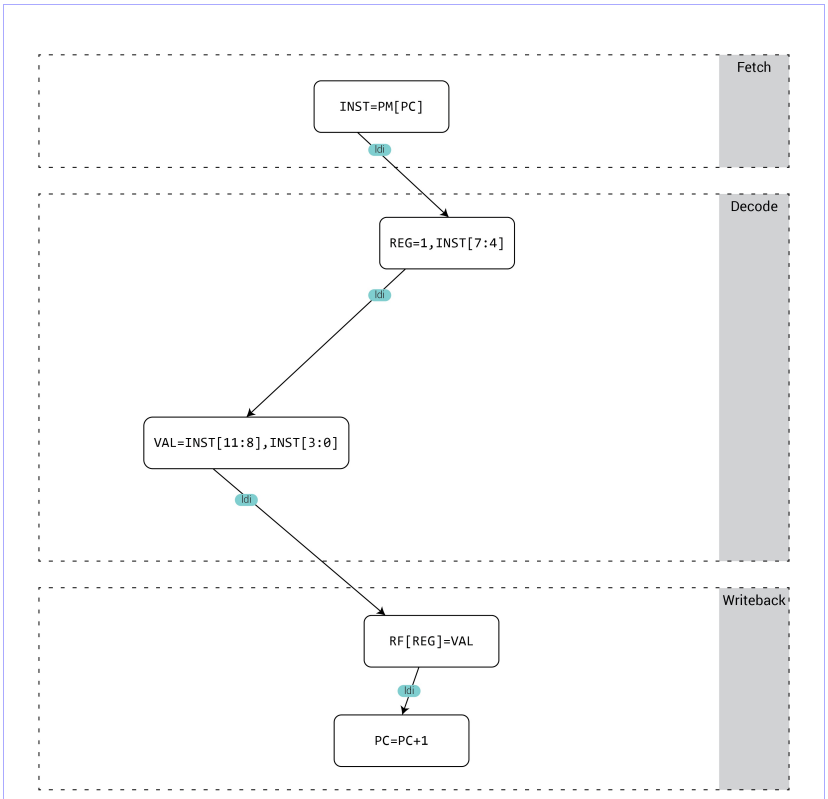

Figure 7.3.1.1:

7.3.2: ld

Now say we wish to add the ability to run both ldi and ld instructions. Let us discuss what needs to be added to each of the stages:

Fetch: We're still just fetching the next instruction in this stage, so nothing changes.

Decode: This is where it gets most interesting. Now, the next state will have to be one of two different things--one will be for if INST was an ldi instruction, and this will proceed as before. The other will be for if it was an ld instruction, to start doing the things that will ultimately perform an ld operation.

ld is a different sort of instruction from ldi: Rather than providing a value and a register number and requiring that the value be written to the register, it provides a register number and an address, and requires that the address in memory be read, and that value stored into the register. So the two pieces we need to split off it are the register number to write to, which we store in REG, and the address to read in data memory, which we store in a new auxiliary memory called ADDR:

```
REG = bits of INST saying which register to write to
ADDR = bits of INST saying which address in data memory to read
```

Execute: Once again, no arithmetic is happening yet, so there is no need to do anything here

Memory: Now we need to do something memory-related for ld instructions, namely, we read from memory. Having acquired the address to read from in ADDR, we can now read the value from data memory at address ADDR. This we need to store somewhere--VAL will do. Thus:

```
VAL = data_memory[ADDR]
```

Writeback: This actually need not change, since for ldi, we would do

```
RF[REG] = VAL
```

But if instead we were doing ld, VAL still contains by this point the value we want to write (it was read from data memory in the previous stage) and REG still contains the address we wish to write to. Thus this will be the operation performed on either an ld or an ldi.

So the new state diagram for a machine that can understand and execute both ldi and ld instructions is:

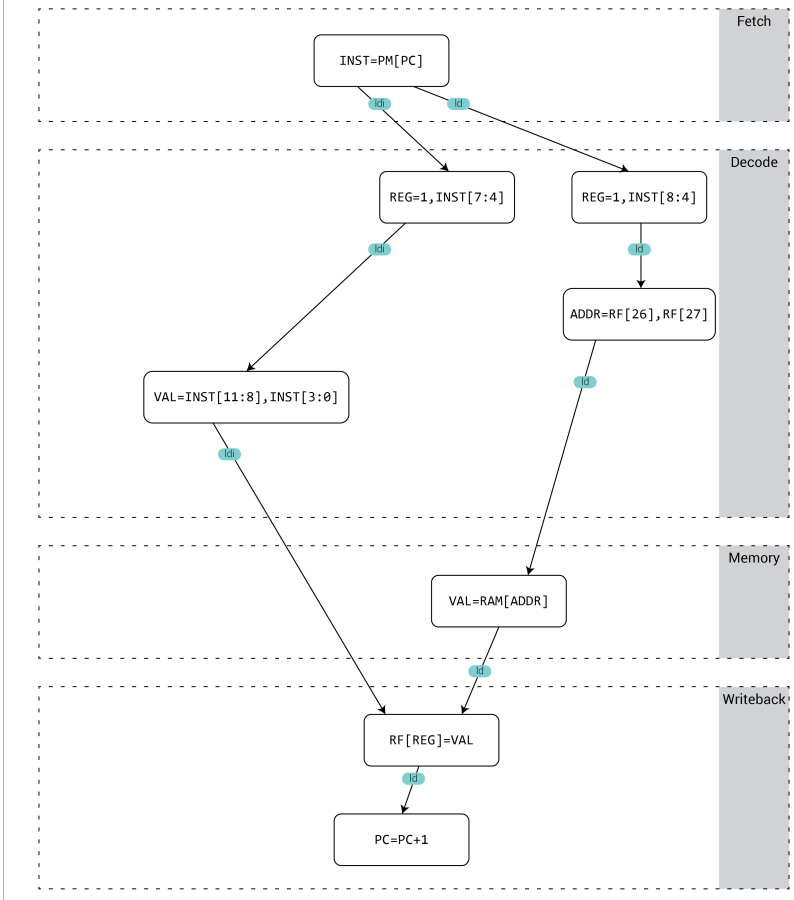

Figure 7.3.2.1:

7.3.3: st

Decode: In the decode stage, we can get the address to write to and the register to read from in exactly the same way, so we actually leave this state intact, and will end up in this state in case of either an ld or an st instruction. However, whereas a ld instruction then just needs to read memory address ADDR and store the value in the register file, st needs to do more: It needs to read out the value from register number REG and store that in memory. So the state branches into the memory read state for ld, and into

```
VAL = RF[REG]
```

for st.

Memory: Now that we have VAL as the value from the register and ADDR as the address to write this to, we can happily send this data off to the data memory, as in the operation:

```
RAM[ADDR] = VAL
```

Writeback: For a st instruction, we do nothing in the writeback phase except increment PC.

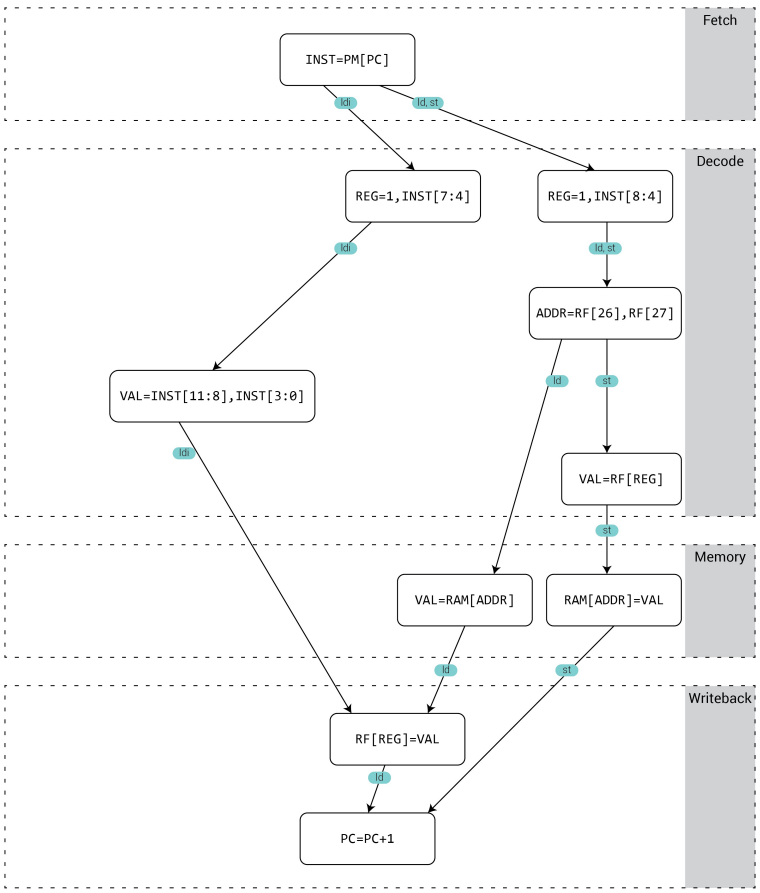

Figure 7.3.3.1:

7.3.4: add, sub, cp

Now we think about how to build in the functionality necessary to execute add instructions. Recall that an add instruction is of the form add reg1, reg2 which should set reg1 = reg1 + reg2 And it is encoded using the format: 000011SRRRRRSSSS where the S bits represent reg2 and the R bits reg1.

Fetch: Once again, we can use the state we have here to get the current instruction.

Decode: For an add instruction, we will need to pull off which two registers we are adding, and then read out the values stored in those registers. The registers we are adding can be read off of the instruction as:

```
REG = INST[8:4]
REG2 = INST[9],INST[3:0]
```

Note that since we are now using two register addresses, we have added a new auxiliary register REG2 to store the second address, since REG is being used to store the first.

So after these two operations, REG and REG2 will contain the numbers of the registers to add (e.g. r5 and r17). Next we need to read the values stored in those registers, now using new auxiliary registers VAL1 and VAL2 (VAL will be used to store the sum):

```
VAL1 = RF[REG]
VAL2 = RF[REG2]
```

However note that these two sets of operations have data dependencies: In order to get the correct

result from VAL = RF[REG] we need first to have run REG = INST[8:4]. So these operations need to come in separate states. Further, the register file only allows reading of one register at a time, so the two reads VAL = RF[REG] and VAL2 = RF[REG2] cannot happen in the same state. So the decode stage for add instructions will consist of three states:

```
REG = INST[8:4]
REG2 = INST[9],INST[3:0]
```

followed by

```
VAL1 = RF[REG]
```

and finally:

```
VAL2 = RF[REG2]
```

Execute: For add instructions, we finally have some computation to do, namely, having read out the values in the two registers REG and REG2 into auxiliary registers VAL1 and VA2 respectively, we now need to add them and store the result somewhere, so we do the operation:

```
VAL = VAL1 + VAL2
```

We also need to update the SREG flags accordingly, so we next perform the operation:

```
Update SREG
```

Memory: An add instruction doesn't touch memory, so this stage is unaffected.

Writeback: Now, exactly as in ldi, we want to write some value to a register, and we have the value stored in VAL and the register number in REG, so we can proceed to the same state as we had before:

```
RF[REG] = VAL
```

Note that almost the exact same flow will execute sub instructions except we need in the execute stage to change the operation to subtraction:

```
VAL = VAL1 - VAL2
```

Allowing for execution of further computational instructions that operate on two registers, such as and, or, eor, etc. is very similar. One other sort of instruction that is similar is cp, which recall does exactly the same thing as sub except it doesn't save anything to the register file. Thus only one new state transition is needed to add cp capabilities to our state machine:

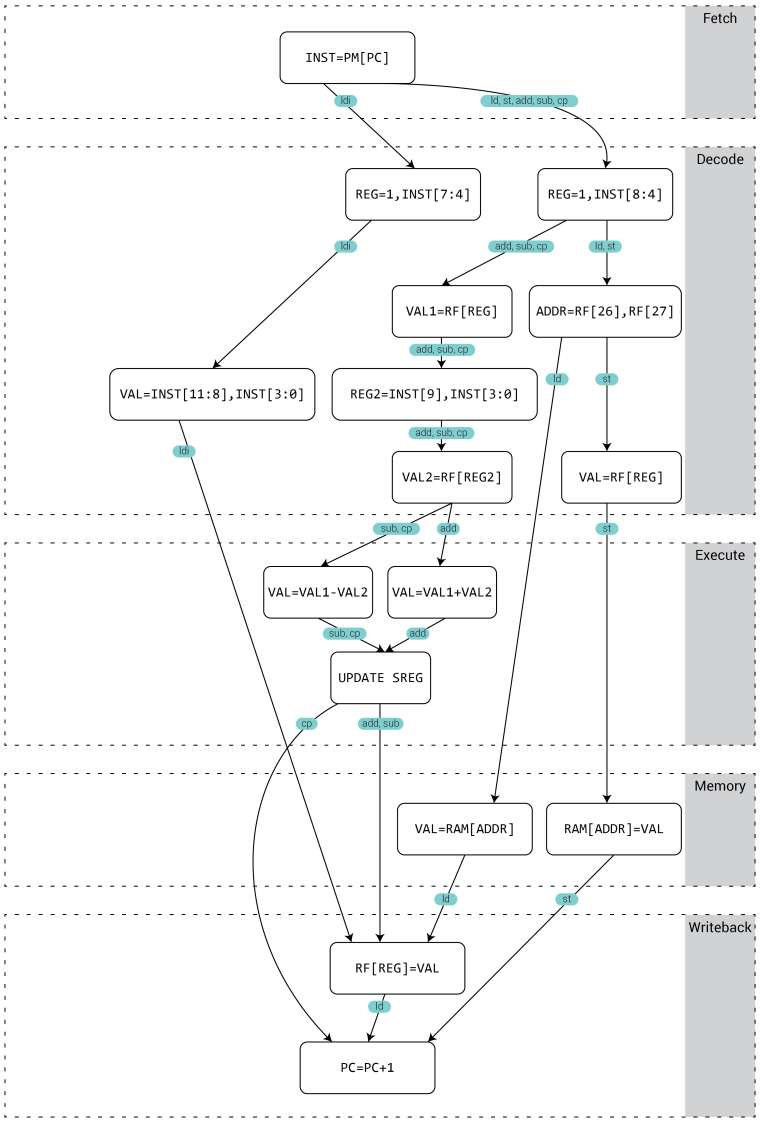

Figure 7.3.4.1:

7.3.5: subi, cpi

We now turn to the cpi and subi instructions, which have formats

```
0011IIIIRRRRIIII
```

and

```
0101IIIIRRRRIIII
```

respectively. Both of these instructions take the register number 1RRRR and subtract from it the immediate value given by the I bits. The only difference between these is that subi actually stores this value, whereas cpi merely updates the SREG.

Decode: Since we want again to take advantage of a pre-existing state--in this case VAL = VAL1 - VAL2--we will use the decode stage to set this up with REG = 1,INST[7:4] followed by VAL1 = RF[REG] and VAL2 = INST[11:8],INST[3:0].

Writeback: In the case of subi, we will go to the usual register-writing state: RF[REG] = VAL and for cpi we skip this state and go straight to PC = PC + 1

In summary, we have the diagram:

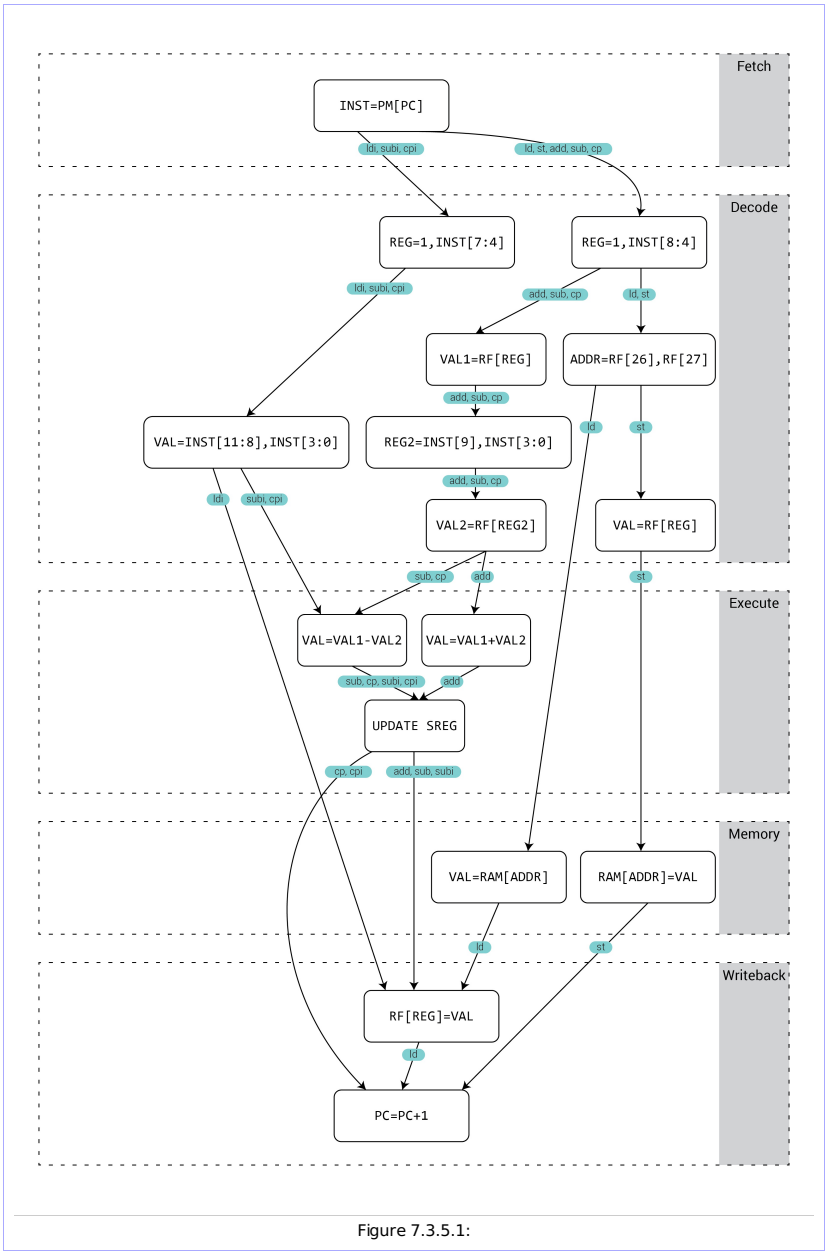

Figure 7.3.5.1:

7.3.6: rjmp

rjmp instructions, we recall, are formatted as:

```
1100IIIIIIIIIIII
```

and have the effect PC = PC + immediate value (encoded as the I bits of the instruction).

Decode: So since we already have a state that performs addition, namely VAL = VAL1 + VAL2, we'll simply reuse this to perform the addition. This means that in the decode stage we need to do VAL1 = PC and VAL2 = the immediate, which we can read off from the instruction as INST[11:0]. So our decode state for rjmp is:

```
VAL1 = PC
VAL2 = INST[11:0]
```

Writeback: But now in the writeback stage, instead of the usual PC incrementing, we just want to set PC = VAL. Thus:

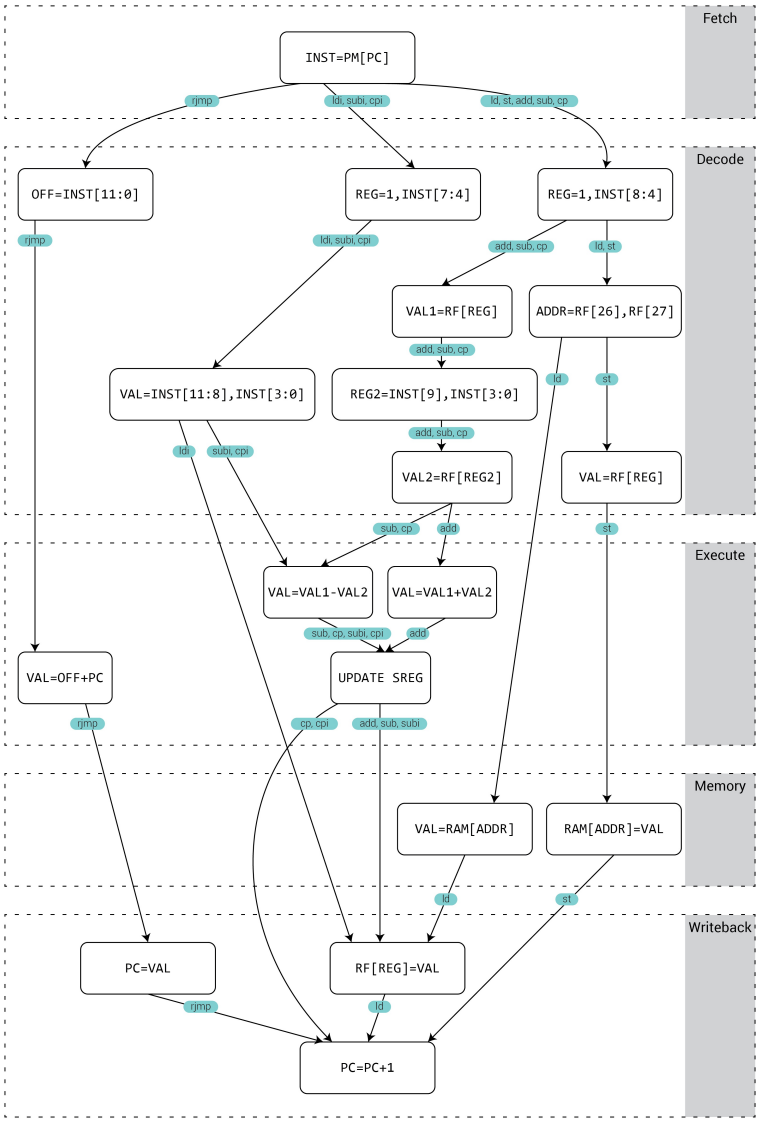

Figure 7.3.6.1:

7.3.7: breq

breq is the final instruction whose implementation we will discuss separately. Its format is:

```
111100IIIIIII001
```

It is meant to execute PC = PC + immediate value in case the Z flag is 1, and PC = PC + 1 otherwise.

Decode: We will use VAL1 = PC, VAL2 = INST[9:3] in the decode stage, and then proceed to the VAL = VAL1 + VAL2 state in execute, and then the PC = VAL stage in writeback. Except now we have a problem, since this will do the same thing as rjmp, whereas we only want that behavior in the event that the Z flag is 1. But the state machine doesn't have SREG as an input at the moment--only INST. There are two solutions: We can change the second operation to something like:

```
VAL2 = {ZZZZZZZ & INST[9:3]}
```

or we can allow SREG to be an input into the state machine, in which case we can branch in decode based on the Z flag. Taking this approach, we get the state diagram:

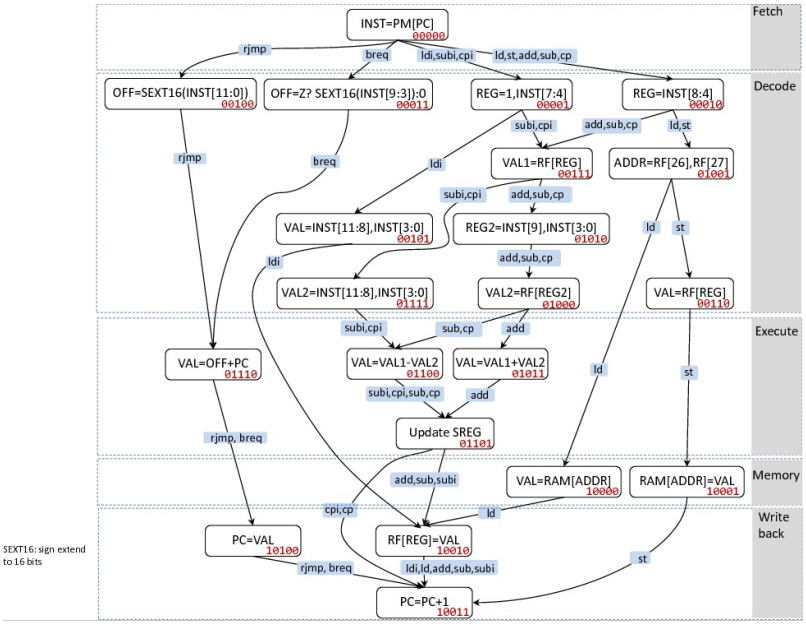

Figure 7.3.7.1: Getting messy now...

7.4: Our microarchitecture as a circuit

Introduce components for register file, aux registers, RAM, PM, and then make circuit for LDI. Note that in the state machine, the address to read from the RF can come from several places. But the RF only has one address input, so which do we connect to it? Trick question--we connect all the possible things that might be used for the address to a multiplexer and then connect that multiplexer's output to the RF's address input.

7.4.1: Circuit components

In 7.1.2 we described the basic circuit components we would be using. Certainly we will need an ALU, some kind of state machine controller, and many multiplexers, but what memories do we need? To answer this, we think back to our description in chapter 5 of the AVR architecture: An AVR computer has the following memories:

- PM: The program memory for storing the instructions. Because this is a read-only memory, its only input is "addr" for specifying an address. It permits a 16-bit address and each address stores 16 bits, so both its address input and data output are 16 bits wide. (TODO add figure)
- RAM: A memory for storing large amounts of data. This memory also allowed 16 bits of address, but each address stores only 8 bits of data, so the address input is 16 bits wide, but the data output is 8 bits wide. Since the RAM is writable, it also has a data input 8 bits wide, and a write-enable input, here shortened to "w_e": (TODO add figure)
- RF: The register file for storing the 32 general-purpose registers. This is a writable memory, but since there are 32 registers, the address input only needs to be 5 bits wide. The data input and output are both 8 bits wide. In addition, it will be convenient to have a separate output for the value denoted X in the ISA--that is, r27:r26. This will be 16 bits wide.
- PC and SREG: The two special-purpose registers for storing the program counter and status, respectively. Since these are registers, they only have data and write_enable as inputs, and only data as output. The PC is 16 bits, while the SREG is 8 bits.
- Auxiliary registers: In addition, according to our state machines, we had some additional registers we would use to store intermediate values along the way to performing the various instruction operations. These will be registers, so will looks exactly like PC and SREG, except for the widths. The auxiliary registers we had were:
  - INST: For storing the current instruction, so of width 16
  - REG: For storing a register number, so of width 5
  - REG2: For storing another register number, so also of width 5
  - VAL: For storing something that would be stored in RAM or a register. Of width 8
  - VAL1: For storing another thing that would be stored in RAM or a register. Of width 8
  - VAL2: For storing another thing that would be stored in RAM or a register. Of width 8
  - OFF: For storing an offset to jump to. Of width 12
  - ADDR: For storing a RAM address, so of width 16

So before we do any thinking about how to hook them up, we know our circuit will have to have these memories, an ALU, and somewhere a state machine controller and some multiplexers. Our challenge now will be to work out how to hook these up to implement the state machine above.

7.4.2: ldi

We'll start by trying to build a circuit that can at least handle ldi instructions. Let's start by collecting the circuit components we'll need: Of course we have to get the instruction, so we'll need the program memory (PM) and the PC. Also we'll be storing something in a register, so we need the register file (RF). Finally, looking back at the state machine for ldi, we'll need the auxiliary registers INST, REG, and VAL. Laying these all out, ready to be connected, we get:

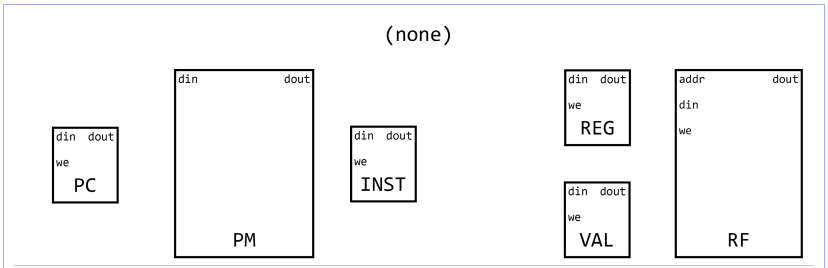

Figure 7.4.2.1:

Further, without thinking too hard, we can see some of the connections straight away: The basic flow of an ldi instruction is that the PC stores the address of the current instruction, so we need PC to feed into PM's addr input. Then, we want to store the current instruction--that is, PM's data output--in INST. So PM's data should be connected to INST's data input.

Now INST has 16 output wires coming out of it. We want to split off 4 of these--namely wires 4-7--to feed into REG's 5-bit input (together with a 1 as the high bit), and we want to split off wires 11-8 and 3-0 to feed into VAL's 8-bit input.

Also, REG should feed into the RF's address input, and VAL should feed into RF's data input.

Finally, we need a way of getting PC + 1 stored into PC, so we feed PC through a circuit component that adds 1 to its input. We feed the output of that component back into PC's data input.

This gives us the basic picture of:

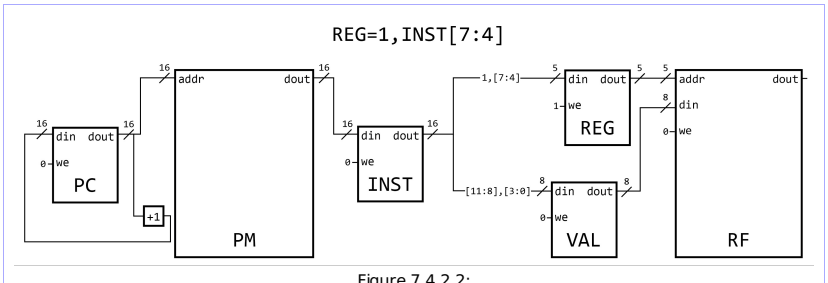

Figure 7.4.2.2:

From here, though, the only things that are left unconnected in this picture are the write-enable wires! Remember that just because we connected e.g. VAL to RF's data input doesn't mean that value will be written to any register--it only will if RF's write-enable input is high. This will provide us exactly the mechanism we need for stepping through the states of the state machine:

INST=PM[PC]:

The first state writes the current instruction at address PC in the program memory to the register INST. In particular, at this point we want none of the registers to be written except for INST. So we can simply set all their write-enable inputs low except for INST's, and we get:

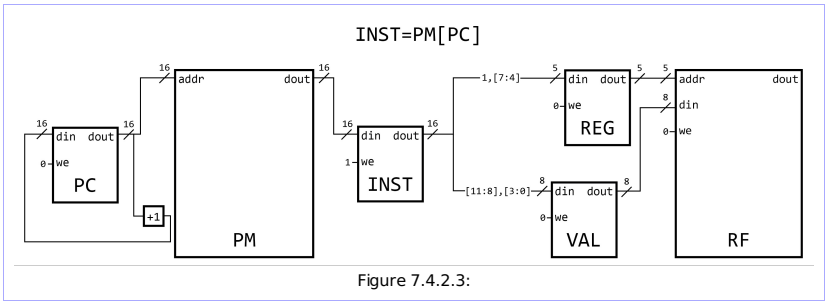

Figure 7.4.2.3:

This will write whatever is in program memory at address PC into the register called INST and will not change anything stored in any other register.

REG=1,INST[7:4]:

The next state sets REG to be some bits of INST (and a high 1). Since these wires are already connected to REG's data input, all we have to do is set REG's write-enable high at this point. Since INST already has the correct value because of the previous state, this will write the correct thing into REG, so at this point, the circuit should look like:

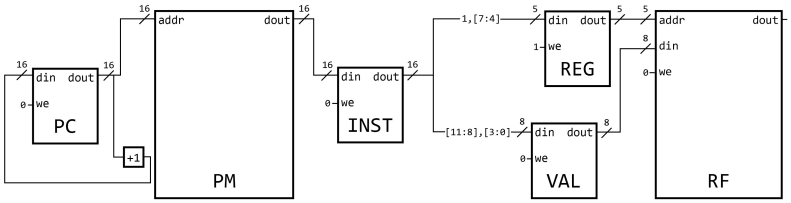

Figure 7.4.2.4:

VAL=INST[11:8],INST[3:0]:

Likewise, since the current instruction is already in INST, the correct thing will be coming into VAL's data input, so for this state we just need to set its write-enable high:

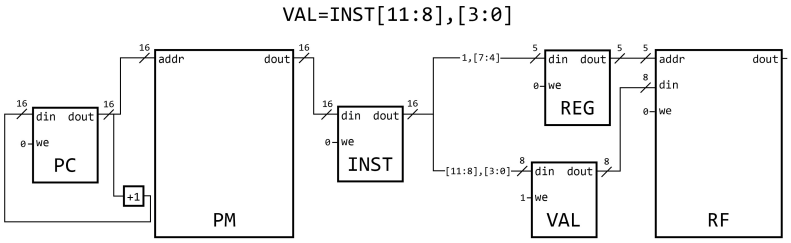

Figure 7.4.2.5:

RF[REG]=VAL:

Now that the value to write is coming out of VAL's data output and the correct address for the register we want to write is coming out of REG's data output, we are ready to set RF's write-enable high to store the desired value in the selected register:

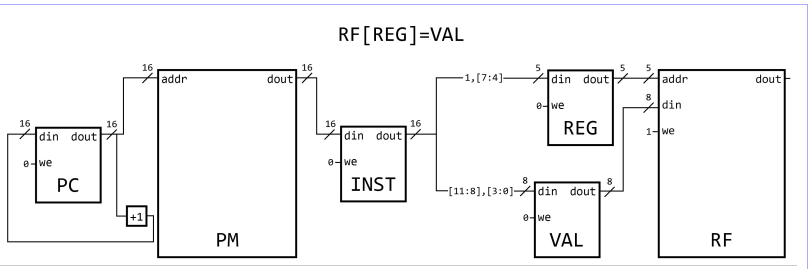

Figure 7.4.2.6:

PC=PC+1:

Finally, to increment the PC, we set PC's write-enable high so that the incoming value of PC+1 gets written.

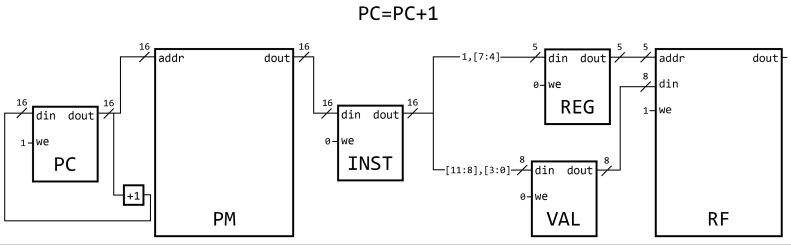

Figure 7.4.2.7:

Putting it all together:

Now that we've seen what each individual state is supposed to look like, now we want to put all these together into a single circuit. The way we do this is by having the state machine controller--that other circuit component we promised to explain later--sitting behind our main circuit and connected to all the write-enable inputs in the circuit, like so:

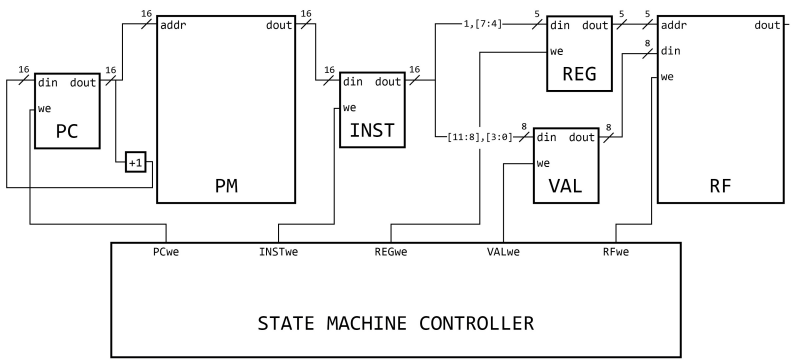

Figure 7.4.2.8:

So what precisely does this state machine controller do? It's only input is inst_in, so it knows what the current instruction is. Using this information, it advances through the states of the state machine automatically, keeping track internally all the while of which state it is currently in.

When it is first turned on, then, it starts in the state INST=PM[PC]. As we saw, to be in this state means that INST_we is high, and the other outputs of the state machine are low. In other words, the circuit receives inputs as in 7.4.2.fig3.

Then, the state machine looks at its input--inst_in--and sees what kind of instruction it is. Since our circuit is only meant to run ldi instructions for the moment, it sees (hopefully) an ldi instruction. According to the state machine diagram, this means it should enter the state . first looks at inst_in. Because this is a machine that only runs ldi instructions, it (hopefully) sees an ldi instruction, and so it enters the state REG=1,INST[7:4]. In practical terms, this means the REG_we output of the state machine controller is high, and the others are low, giving a situation exactly as depicted in 7.4.2.fig4.

As the only thing that changed as a result of the previous state was REG, in particular INST has remained the same, so the input is still the previous ldi instruction, so we proceed to the next state: VAL=INST[11:8],INST[3:0]. Again, what it means to be in this state is that the VAL_we output of the state machine controller is high and its other outputs are low. This replicates the situation in 7.4.2.fig5.

Next, the state machine controller enters the state RF[REG]=VAL. All this means, once again, is that the RF_we output is high and the others are low, meaning that the situation is as depicted in 7.4.2.fig6.

Finally, the state PC=PC+1 is entered, meaning that the PC_we output is high and the others are low as in 7.4.2.fig7. This causes PC to get overwritten with whatever's coming in on its din input--in this case PC+1. This causes the incremented PC to get stored in the PC register, and also to go out on the PC register's dout output. This in turn goes to the PM and causes it to spit out whatever instruction was stored at this new address on the PM's dout output. This goes to INST's din input, ready to be stored. Thus when the state machine advances to the next state which is INST=PM[PC] and once again sets INST_we high, the next instruction will be stored in the INST register and therefore will be the new input to the state machine, starting another cycle of the same process on the next stored instruction.

What we've seen through all of this is that the state machine controller is just a circuit that steps through the states of the state machine according to its input, except that for each state, we have to specify what it means to be in that state in terms of the state machine controller circuit's outputs. In this case, we can specify its behavior entirely by saying, for each state, what each output should be:

	PC_we	INST_we	REG_we	VAL_we	RF_we
INST=PM[PC]	0	1	0	0	0
REG=1,INST[7:4]	0	0	1	0	0
VAL=INST[11:8],INST[3:0]	0	0	0	1	0
RF[REG]=VAL	0	0	0	0	1
PC=PC+1	1	0	0	0	0

So the state machine diagram determines the states that the state machine goes through, and this table determines what output values of the state machine circuit correspond to the various states. The end result is that the state machine diagram together with this table define the behavior of the state machine circuit completely.

7.4.3: ld

Adding one additional instruction to our circuit will be the hardest step. Once we see how to do this, adding the remaining instructions will follow easily. So let us walk through the states as we did for ldi and see what issues arise:

INST=PM[PC]:

This is actually exactly the same, so no changes need to be made at this point.

REG=INST[8:4]:

Now we see our first real problem: To execute an ldi instruction, we need to feed 1,INST[7:4] into REG, but to execute an ld instruction, we need to feed REG the value INST[8:4]. But REG only has one data input, so which of these do we connect?

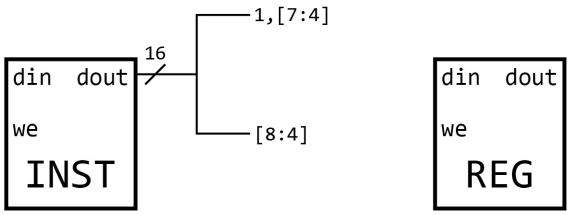

Figure 7.4.3.1:

The answer is that we connect both of these inputs to a mux, and connect that mux's output to REG's data input like so:

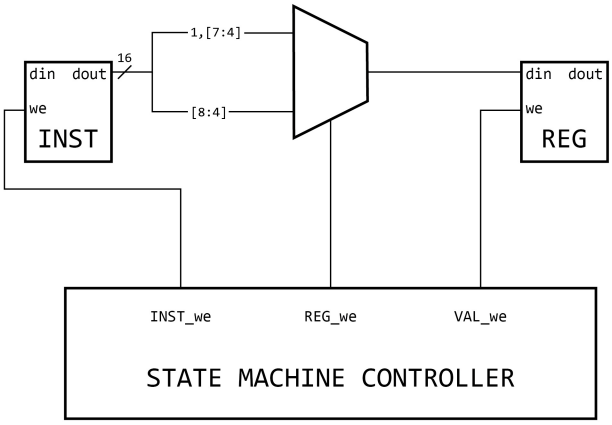

Figure 7.4.3.2:

This way, just as the all-powerful state machine controller got to choose which register was being written to at each time, now it also gets also to choose at any given time which value is written to the register! But how does it decide? It needs to know the current instruction, so we add an input to the state machine controller, into which we send INST--the current instruction.

ADDR=X:

Next, we need to set the address equal to the value X, which is handily an output from the RF already. So we just connect RF's X output to ADDR's data input:

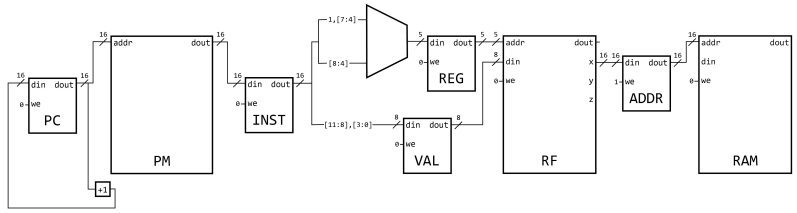

Figure 7.4.3.3:

VAL=RAM[ADDR]:

Now that we have fed ADDR into RAM's address input and the value at that address in RAM will come out of its data output. In this state, then, we only need to connect RAM's data output to VAL's data input and we're home free. Except wait--VAL already has a data input for the event of an ldi instruction! So we repeat exactly the muxing strategy from before, and we come up with the following:

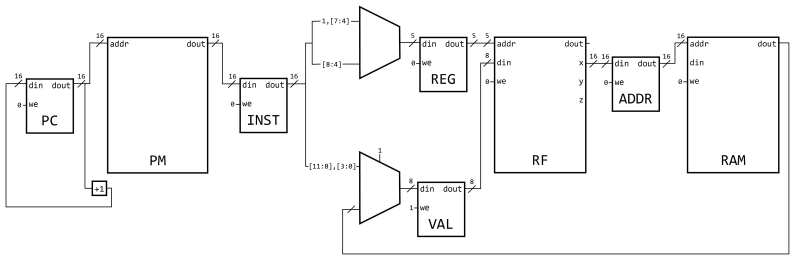

Figure 7.4.3.4:

RF[REG]=VAL:

This state is actually identical to what happens for ldi, so we can use the same exact inputs as before:

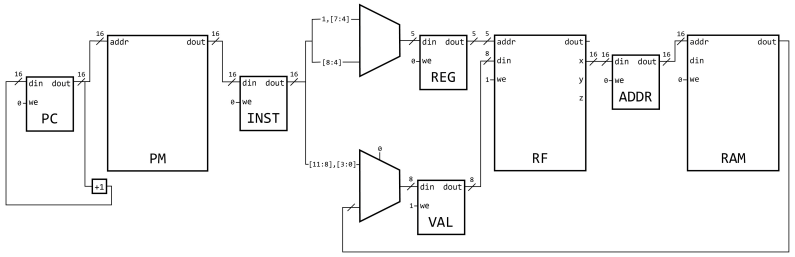

Figure 7.4.3.5:

Putting it all together:

Once again, to advance through these states, we simply take all the inputs that change depending on which state we're in--in this case, that's all write-enable inputs and all mux select lines--and connect them all to the overarching state machine controller. Since the state machine controller needs to know which instruction is being executed in order to decide what state to enter next (since now the instruction might either be an ldi or an ld, whereas in the previous section the only possibility was an ldi), it now needs to receive INST's data out as its input. All told, we get the following circuit diagram for a computer that can execute both ldi and ld instructions:

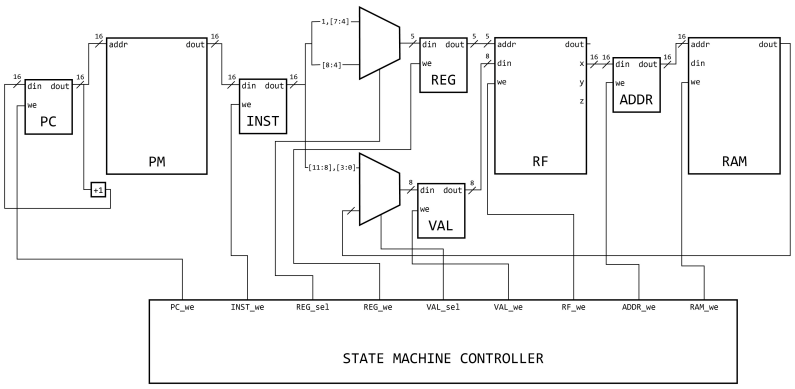

Figure 7.4.3.6:

Once again, we can describe the state machine controller's behavior by reference to the appropriate state machine diagram--in this case 7.3.2.fig1--and a table explaining what the various outputs do in each state, which we supply here:

	PC_we	INST_we	REG_we	REG_sel	VAL_we	VAL_sel	RF_we	ADDR_we	RAM_we
INST=PM[PC]	0	1	0		0		0	0	0
REG=1,INST[7:4]	0	0	1	0	0		0	0	0
REG=INST[8:4]	0	0	1	1	0		0	0	0
ADDR=X	0	0	0		0		0	1	0
VAL=INST[11:8],INST[3:0]	0	0	0		1	0	0	0	0
VAL=RAM[ADDR]	0	0	0		1	1	0	0	0
RF[REG]=VAL	0	0	0		0		1	0	0

PC=PC+1	1	0	0		0		0	0	0

We have left some of these entries blank when they don't matter: For instance, if REG_we is 0, then it doesn't matter what is coming into the din input of REG, as it won't actually be written. Since this input is chosen by REG_sel, the actual value of REG_sel doesn't matter. Of course, it is a wire in a digital circuit, so it will still have either current going through it or not--that is, it will be a 0 or 1. By leaving the slot blank, we are just indicating that it doesn't matter which. For actually building the circuit, we could choose, for instance, that whenever an output doesn't matter, we'll make it 0.

To understand how this works, let us walk through a few steps of this circuit's life:

The state machine always starts in the state INST=PM[PC]. That is, INST_we is high and all the write-enable outputs are low. This means that whatever's coming out of PM's data output is going to be written into INST. This new instruction then appears on INST's dout output, and is therefore also fed to the state machine's inst_in input. Suppose the instruction is the number 1001000101001100. According to the state machine's built-in table of encodings, this is an ld instruction, so it transitions to the state REG=INST[8:4]

In this state, REG_sel is 1, meaning that the mux outputs whatever's coming in on its input 1. In this case, that's bits 8-4 of INST. Specifically, the bits 10100 (representing the number 20). These are fed into REG's data input din, and as REG_we is also 1, they are in fact stored in REG. INST is of course unchanged through this process, and so the input to the state machine is also unchanged, so it is still an ld instruction, and therefore the state machine transitions to the next state: ADDR=X

In the ADDR=X state, we see that only ADDR_we is high, so ADDR stores whatever is coming into its data input. In this case, that's whatever is coming out of RF's X output--i.e. the value of the special pointer register X.

The next state is VAL=RAM[ADDR], in which we see VAL_we = 1 and VAL_sel = 1. VAL_sel being high means that input 1 of the mux feeding VAL's data input passes through. Coming into this input was RAM's data output, so VAL's data input receives the output of RAM. Specifically, as RAM's addr input receives its value from ADDR, VAL will receive whatever was stored in RAM at address ADDR. Since VAL_we is also high, this value now gets stored into VAL.

The next state is RF[REG]=VAL, in which we see RF_we = 1, and so the input to the RF's data in, which comes from VAL, is written to address coming in on the RF's addr input--which in this case is coming from REG.

Finally, in the PC=PC+1 state, PC_we is high, so the value of PC+1 that is always coming in on PC's data input is finally allowed to be written, and PC is incremented.

Then we return to the state INST=PM[PC]. Since the PC has been incremented, a new value comes into PM's addr input, causing a new instruction to come out and into INST, and therefore into the state machine's input, and the cycle continues with the next instruction.

7.4.4: Everything else

At this point, we've actually seen all the tricks there are to designing a state machine as a circuit: Every write-enable input is connected to the state machine controller, as is every mux's select line. To execute a state that writes a specified value to a specified memory, the state machine will set only that memory's write-enable high and every other one low, and if that memory's data and address input lines are coming from muxes, the state machine controller will set those muxes' select lines to give the memory the correct inputs.

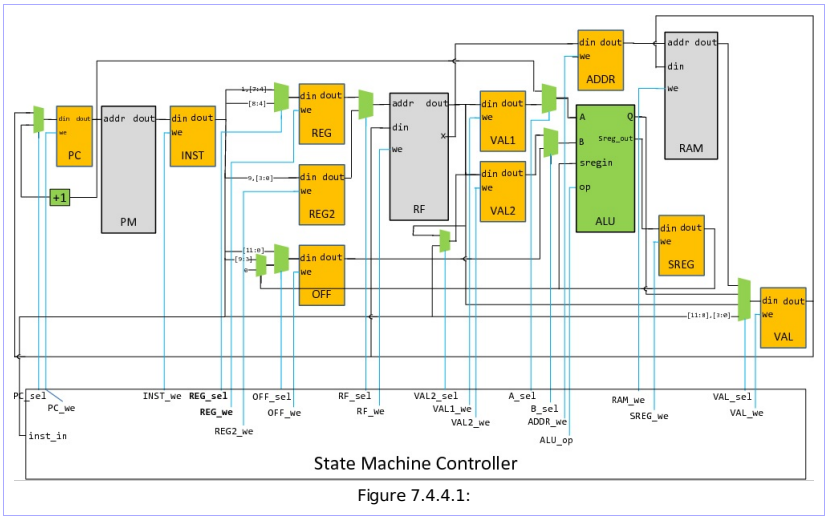

Figure 7.4.4.1:

As this is intended to represent a machine that handles all the instructions we're targeting in this chapter, the state machine should implement the full state machine from 7.3.7.fig1. However to describe its behavior we need not only describe how it transitions between states, but also what outputs it gives in each state. To this end, we present these in the usual tabular format:

State number	State name	PC_sel	PC_we	INST_we	REG_sel	REG_we	OFF_we	OFF_sel	REG2_we	RF_sel	RF_we	VAL1_we	VA

00000	INST = PM[PC]		0	1		0	0		0		0	0	0
00001	REG = 1,INST[7:4]		0	0	0	1	0		0		0	0	0
00010	REG = INST[8:4]		0	0	1	1	0		0		0	0	0
00011	OFF = Z? SEXT16(INST[9:3]):0		0	0		0	1	1	0		0	0	0
00100	OFF = SEXT16(INST[11:0])		0	0		0	1	0	0		0	0	0
00101	VAL = INST[11:8],INST[3:0]		0	0		0	0		0		0	0	0
00110	VAL = RF[REG]		0	0		0	0		0	0	0	0	0
00111	VAL1 = RF[REG]		0	0		0	0		0	0	0	1	0
01000	VAL2 = RF[REG2]		0	0		0	0		0	1	0	0	1
01001	ADDR = X		0	0		0	0		0		0	0	0
01010	REG2 = INST[9],INST[3:0]		0	0		0	0		1		0	0	0
01011	VAL = VAL1 + VAL2		0	0		0	0		0		0	0	0
01100	VAL = VAL1 - VAL2		0	0		0	0		0		0	0	0
01101	Update SREG		0	0		0	0		0		0	0	0
01110	VAL = OFF+PC		0	0		0	0		0		0	0	0
01111	VAL2 = INST[11:8],INST[3:0]		0	0		0	0		0		0	0	1
10000	VAL = RAM[ADDR]		0	0		0	0		0		0	0	0
10001	RAM[ADDR] = VAL		0	0		0	0		0		0	0	0
10010	RF[REG] = VAL		0	0		0	0		0	0	1	0	0
10011	PC = PC+1	1	1	0		0	0		0		0	0	0
10100	PC = VAL	0	1	0		0	0		0		0	0	0

7.5: Exercises

7.1: What does the microarchitecture provide?

7.2: A microarchitecture is a(n) _____ of the _____.

1. outline, computer's high-level programming
2. processor mainly used in mobile devices, current generation of mobile devices
3. implementation, ISA's interface

7.3: As described in the text, the implementation of a microarchitecture happens in which two steps?

1. coding in high-level language, translating (compiling) to assembly language
2. describe using a state machine, create using circuit components
3. gathering physical components, soldering them together on a circuit board
4. assembling the ISA instructions, creating logic circuits from transistors

7.4: What three main parts are there to a state machine?

7.5: List the five stages of execution among all states and briefly explain what they do.

7.6: Circuits are fundamentally _____ connecting _____.

1. components, loops
2. wires, components
3. transistors, logic gates

7.7: What four components do we use in our state diagram discussed in 7.2.2?

7.8: For this question, refer to the last state machine example from this section (the one with the lock that affects only whether you can turn the knob). The following flow-chart specifies a start state and a series of inputs. Your job is to fill in the transition and state info for each of these consecutive inputs.

```
State (start): OPEN, UNLOCKED
Input: close
Transition: move to ___?___ state
    ||
    \/
State: ___?___
Input: lock
```

```
Transition: ___?___
    ||
    \/
State: ___?___
Input: unlock
Transition: ___?___
    ||
    \/
State: ___?___
Input: open
Transition: ___?___
    ||
    \/
State: ___?___
Input: lock
Transition: ___?___
    ||
    \/
State: ___?___
Input: close
Transition: ___?___
    ||
    \/
State (end): ___?___
```

7.9: Name the 5 stages of execution.

7.10: Which of these stages would take an instruction in the form 1100IIIIIIIIIIII and determine that this is an rjmp instruction and store the immediate value "IIIIIIIIIII" in an auxiliary register for later use?

7.11: Which of these stages would then use the "IIIIIIIIIII" value that was stored to update the PC?

7.12: What is the difference between analog and digital circuits?

7.13: If the input values range from 0-7, how many input wires (i.e. what input "width") does the circuit component require to handle this range? What range of inputs can be used on a circuit component with input width of 5?

7.14: What is the "we" input in the figures from this section? How does this input affect the circuit component when it is high vs. when it is low?

7.15: Both the "carry" input and the "C" output of the ALU signify operations that result in a carry. The "carry" input is high if the ____ operation resulted in a carry. The "C" output is high if the ____ operation resulted in a carry.

1. previous; current
2. current; previous

7.16: What is the purpose of the state diagram of Section 7.3 with respect to translating ISA instructions?

7.17: Briefly give a walkthrough of the subi instruction and how it can be given as a state machine representation.

7.18: Describe what happens during each of the 5 stages of execution for the following instruction: 0000111100111100.

7.19: What kinds of memory and registers (including auxiliary registers) do we need to implement our state machine into a physical circuit?

7.20: What are the write-enables and multiplexer we've discussed, and what component controls them in a physical circuit?