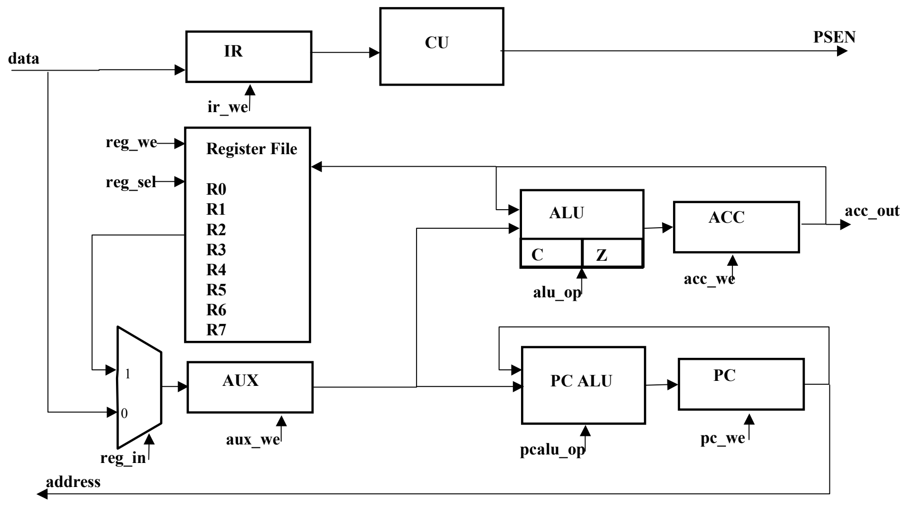
Wimp51 Microprocessor

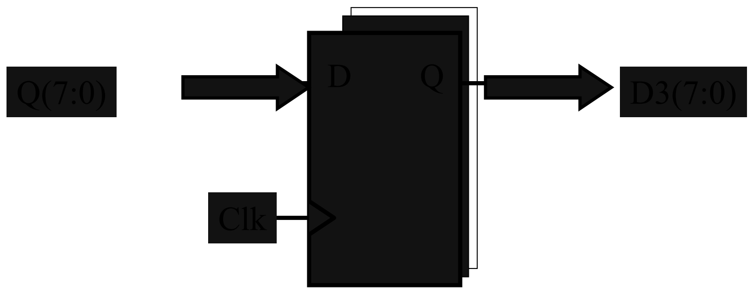


This is the Wimp51 block diagram. Registers include the eight-bit ACC, IR, PC, AUX, and R0-R7 registers; the single-bit C and Z flags; and the two-bit current state register inside the control unit. IR is the instruction register, ACC is the accumulator, PC is the program counter, and AUX is the auxiliary register. The rest of the logic is combinational and includes the AUX input multiplexor, control unit next state logic (CU), and the two arithmetic and logic units (ALU and PCALU). The data bus and address bus are connected to external memory. The address bus is driven by the Wimp51. When the active-low signal PSEN goes low, the external RAM, ROM, or I/O will drive the data bus with the data from the appropriate address.

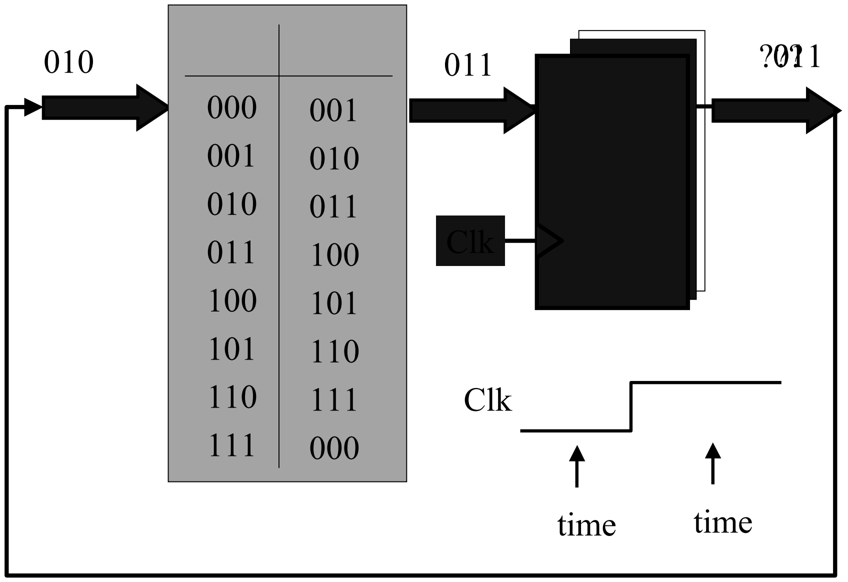
This block diagram did not just appear out of thin air. It is a fairly typical example of an accumulator based architecture and is normally obtained after careful study of the proposed instruction set and several iterations. A course in computer organization typically spends a fair amount of time developing block diagrams like this and comparing various alternative designs.

Register = array of flip flops

- Stack 8 flip flops to get 8 bit data register
- Strobe Clk to get D3 <= Q;

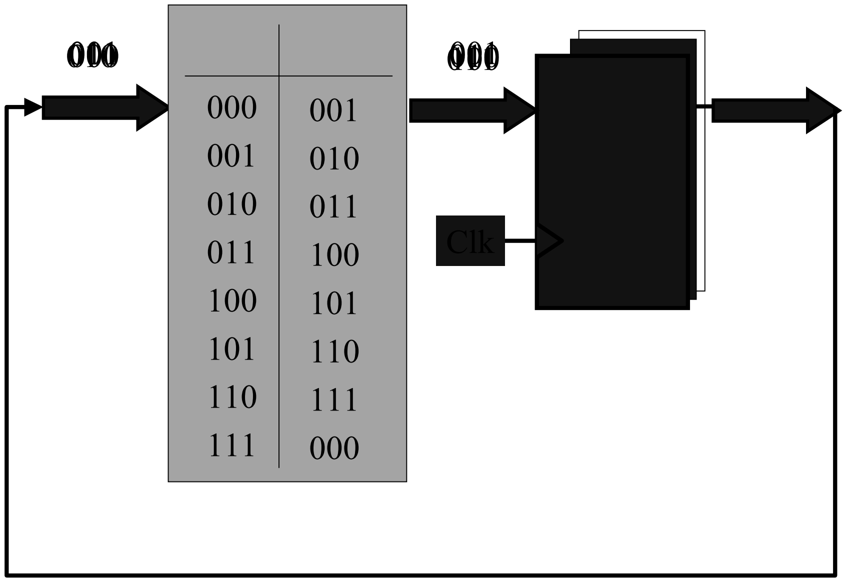


Register + Logic = State machine



000	001
001	010
010	011
011	100
100	101
101	110
110	111
111	000

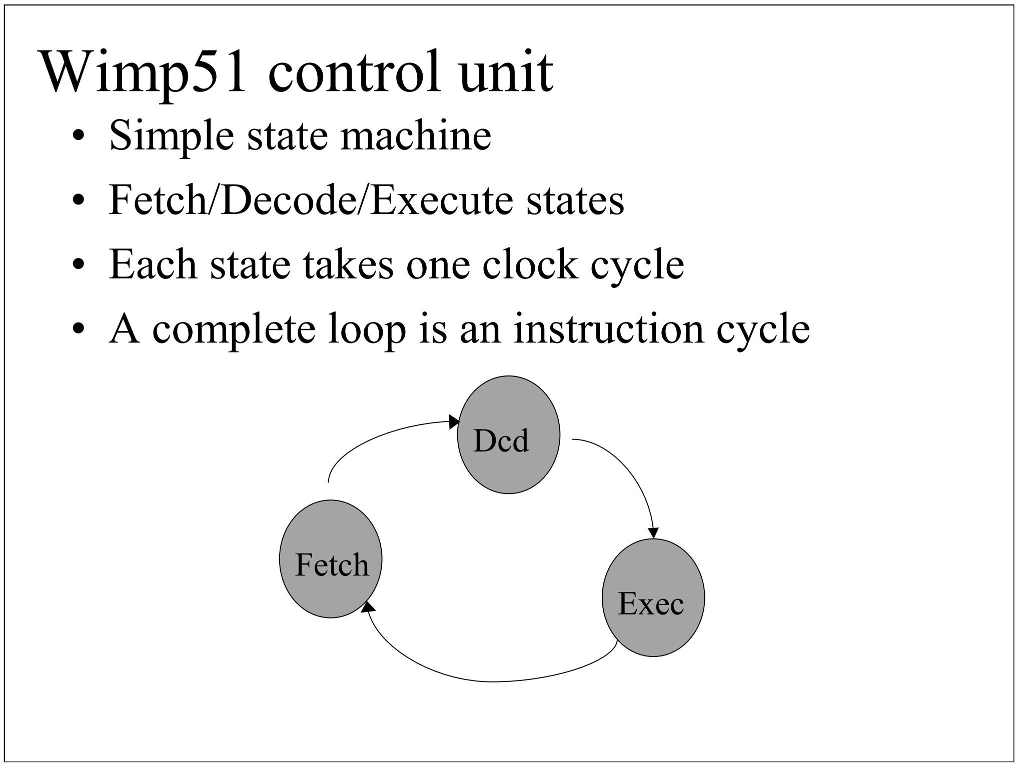
A counter!



000	001
001	010
010	011
011	100
100	101
101	110
110	111
111	000

Wimp51 subsystems

- PC is an eight bit counter, 0 to 255
- PC can also be loaded from the abus
- ACC and ALU form a ‘register with logic’ block
- IR and memory form a ‘register with logic’ block
- Control Unit (CU) is a 2 bit counter, 0 to 2
- Datapath = IR, AUX, PC, ACC, Regfile, ALU, PCALU
  - 12 registers plus Logic (ALU, PCALU)



Wimp51 control unit

- Simple state machine
- Fetch/Decode/Execute states
- Each state takes one clock cycle
- A complete loop is an instruction cycle

$$\begin{tikzcd}
 & \text{Dcd} \arrow[dr, bend left] & \\
\text{Fetch} \arrow[ur, bend left] & & \text{Exec} \arrow[ll, bend left]
\end{tikzcd}$$

Each state lasts for one clock cycle. During the fetch state the current instruction is fetched from code memory and loaded into the instruction register. During the decode state the instruction is decoded and any required operands are fetched from memory or from the register file. During the execute state the instruction is executed. A result may be stored in a register or the accumulator updated. All three states constitute what is often called an instruction cycle.

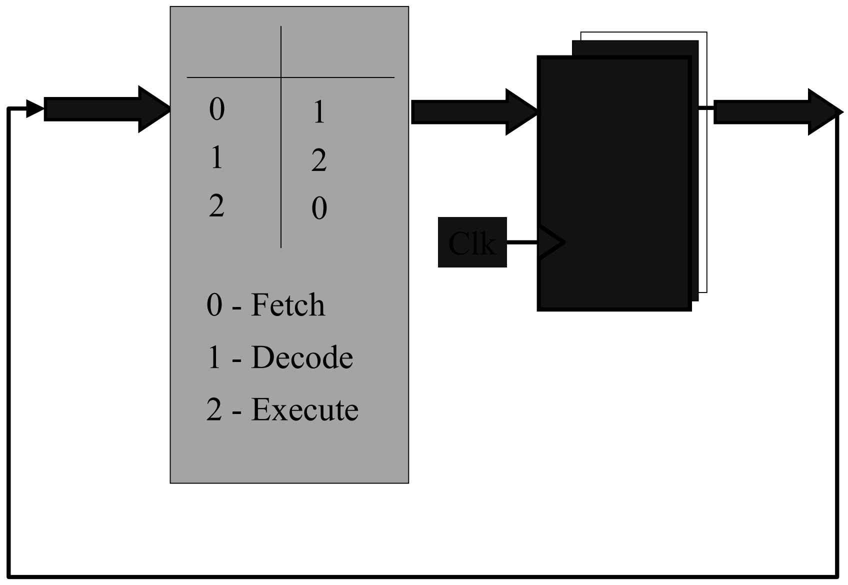
A processor designer is interested in reducing the number of states and in how fast the state machine can be clocked since both determine the speed of the processor. It's no good to have a very high clock rate if it takes a large number of states per instruction.

Questions:

1. Draw a timing diagram showing a clock, the current state, and next state for at least two instruction cycles.

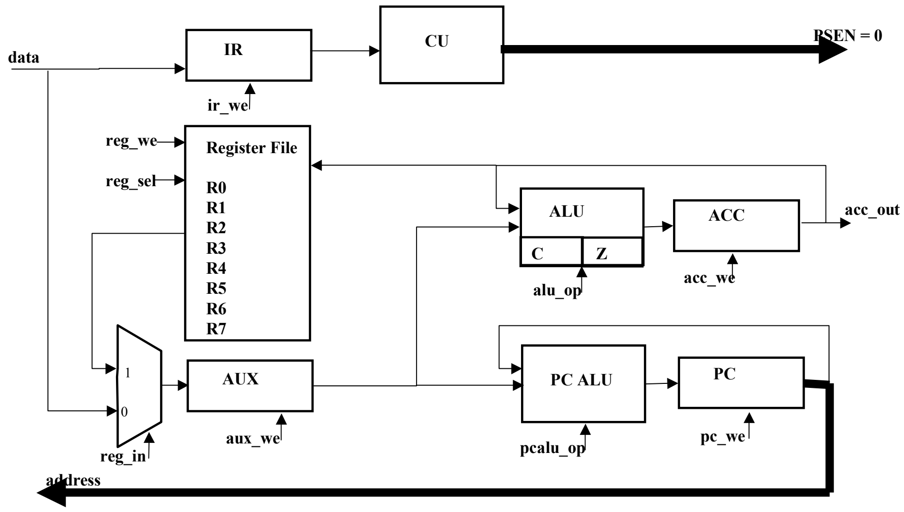
2. Describe what needs to be done to implement the state machine controller in hardware.

Wimp51 control unit



Here's how to implement the CU. This isn't the whole picture though. It only includes the state sequencing. You need to add outputs to enable various transfers such as a PC clock enable, AUX multiplexor control signal, ALU function code, and the like.

Fetch cycle step 1: addr <= PC

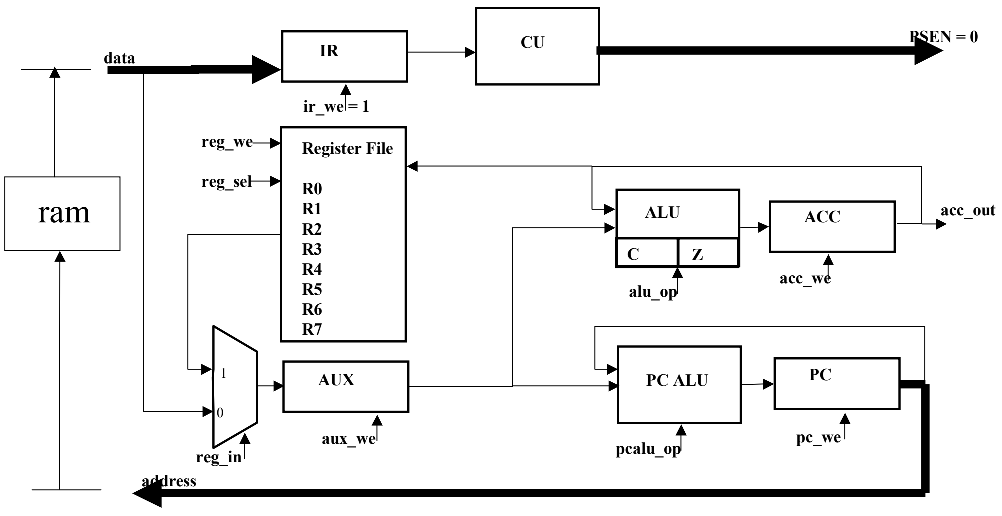


When the CU state machine is in the fetch state, the address bus (addr) is driven with the contents of the PC. Since the PC is only 8 bits wide and the address bus is 15 bits, an additional 8 bits of ‘0’ is used to pad out the address (although it is not shown here). The address bus is 15 bits wide to accommodate a 32k byte SRAM chip for external memory. In fact a variety of schemes could be used to derive the address from the PC. The main idea is that the PC is used to generate the memory address during the fetch cycle.

Questions:

1. What is the range of code addresses?
2. What is the range of memory addresses in hexadecimal?

Fetch cycle step 2: IR<= data

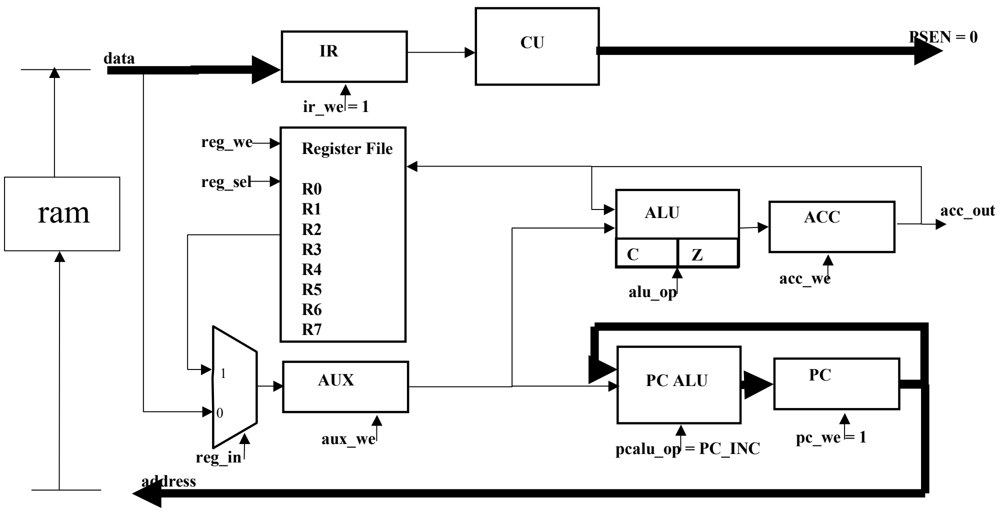


After tacc (ram access time) memory drives the data bus with the instruction at location addr. Sometimes this is abbreviated as IR= C(addr). Note that the data is simply made available at the input of IR. Since IR is just 8 edge triggered D flip flops we can call the input ir_d (ir under d). No latching of the data occurs until the next rising edge of the clock. The rising edge of the clock terminates the fetch cycle and begins the decode cycle.

Question:

A D-FF will latch its input whenever a rising clock edge occurs. We only want IR to latch the data bus at the end of the fetch cycle and not during the other two states. How can we cause IR to change at the end of the decode cycle and keep it the same during the other two cycles? This is an important technique which we'll want to use on all the other machine registers.

Fetch cycle: PC<= PC+1

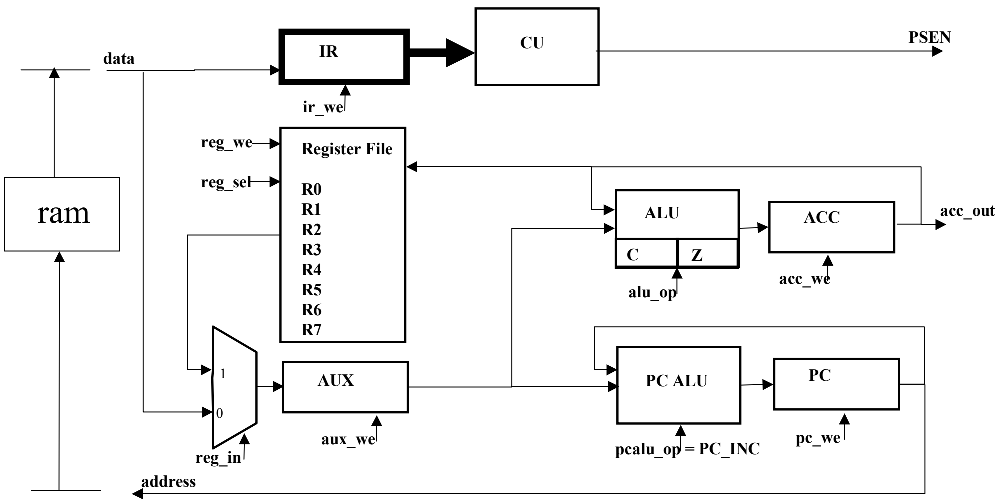


While the next instruction is being fetched, the PC ALU control signal is set to PC_INC so that the PC will get incremented at the end of the fetch cycle. Note that like the IR register, PC+1 is only presented to the input of the PC flip flops and is not latched until the next rising clock edge.

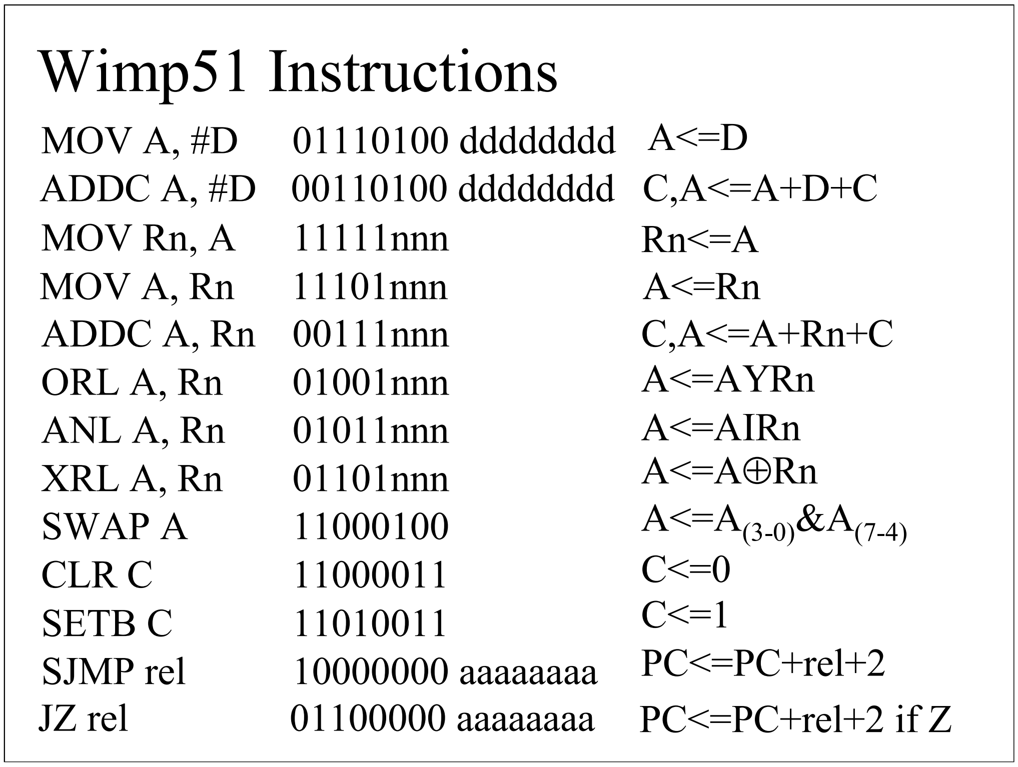
PC is essentially an 8 bit binary counter that can alternatively be added to the value of the AUX output bus. The PC counter is synchronous and thus only changes on rising clock edges. If pcalu_op is set to PC_ADD instead of PC_INC, the PC is added to AUX to form a new address. This occurs during the execute cycle of a JZ or SJMP instruction. If pcalu_op is set to PC_INC the PC is incremented instead. Does this mean that the PC is incremented 3 times per instruction cycle? No! There's actually a control signal pc_we which causes the current value in the PC to be latched. The same thing is true for other registers. If the register is to stay the same then it is simply loaded with its current value. Thus all registers are clocked on every rising clock edge. This is called a fully synchronous design which is the preferred technique for modern ASICs (application specific integrated circuits) and other digital chips.

Decode cycle:

•Decode current instruction in IR



The rising clock edge which causes the CU state machine to transition from the fetch state to the decode state also latches the new instruction present on the data bus into the instruction register (IR). The instruction decode logic responds to the changes in the instruction register by asserting and deasserting various control signals such as *PSEN* and *alu_op*. The operations which occur are dependent on the current instruction in IR and can include fetching a second byte from code memory. We will look at some of these in the next few slides. First let's look at the instruction set for the Wimp51.



Wimp51 Instructions

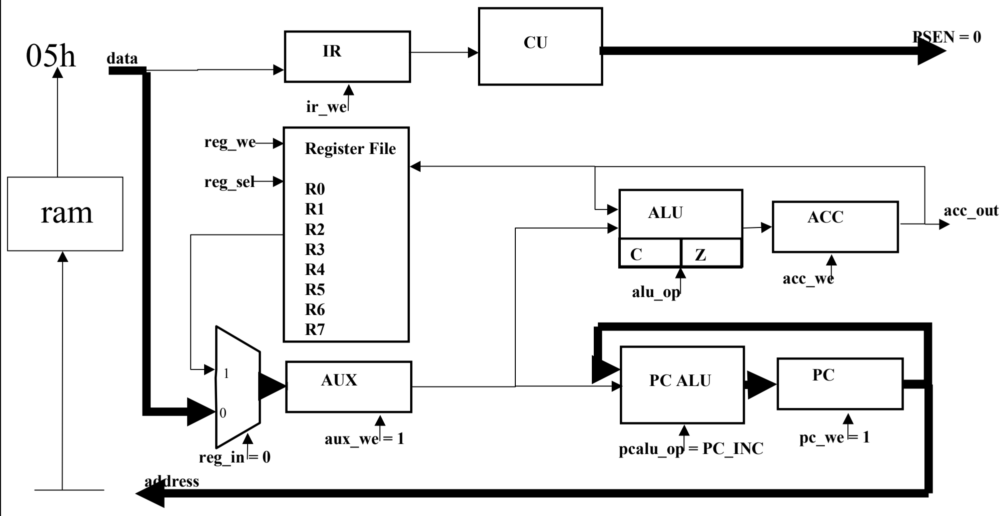
MOV A, #D	01110100 dddddddd	A<=D
ADDC A, #D	00110100 dddddddd	C,A<=A+D+C
MOV Rn, A	11111nnn	Rn<=A
MOV A, Rn	11101nnn	A<=Rn
ADDC A, Rn	00111nnn	C,A<=A+Rn+C
ORL A, Rn	01001nnn	A<=AYRn
ANL A, Rn	01011nnn	A<=AIRn
XRL A, Rn	01101nnn	A<=A⊕Rn
SWAP A	11000100	A<=$A_{(3-0)}$&$A_{(7-4)}$
CLR C	11000011	C<=0
SETB C	11010011	C<=1
SJMP rel	10000000 aaaaaaaa	PC<=PC+rel+2
JZ rel	01100000 aaaaaaaa	PC<=PC+rel+2 if Z

The Wimp51 has a primary 8 bit opcode. Opcodes 74h and 34h are used for load and add immediate instructions. The next eight bits of the instruction are used as an immediate operand so these are called *immediate mode* instructions. Opcodes F8-FF are called *register destination* instructions since they use the lower three bits of the opcode to select a register to write the result into. Opcodes 38-3F, 48-4F, 58-5F, 68-6F, and E8-EF are all *register source* instructions since the lower three bits of the opcode select a register to read the second operand from. Opcodes C3, C4, and D3 all operate on a single register and do not require any operand at all. The two jump instructions, opcodes 80h and 60h, require a second byte to be fetched from memory and added to the program counter. This second byte is called the *relative address*.

As you can see, there are many different kinds of instructions in the Wimp51 instruction set. Even more complex is the actual 8051 instruction set with a few more addressing modes and 98 more instructions.

Decode cycle: immediate source

MOV A, #05h

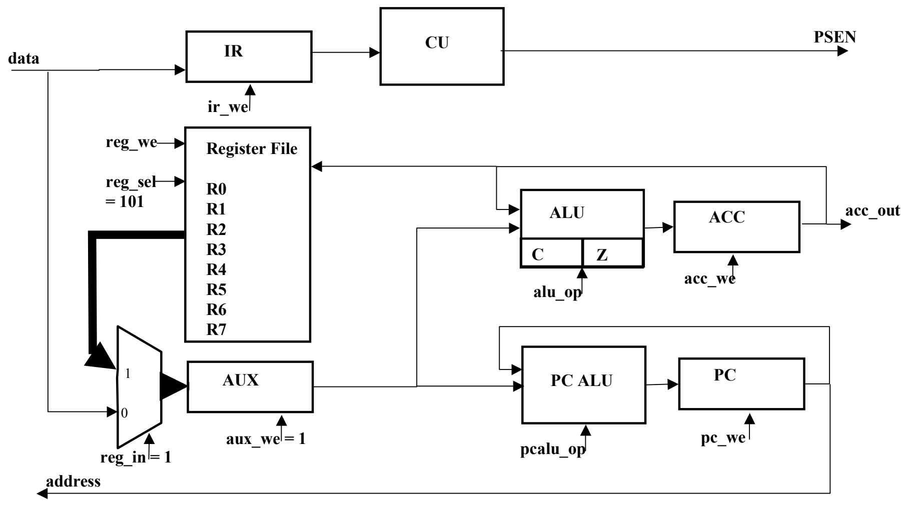


Here we examine the operations during the decode cycle while the instruction register contains a MOV A, #05h. This instruction is encoded as B"0111_0100" or 74h. When the control unit decodes this instruction, it will drive PSEN low to cause a second byte to be fetched from memory. The AUX multiplexor control line is set to 0 to cause the AUX register to latch the data bus on the next rising clock edge. In this case, the second byte is B"0000_0101" or 05h.

Immediate source instructions for the Wimp51 include MOV A, #data; ADDC A, #data; SJMP rel; and JZ rel. For all four of these instructions, the second byte of the instruction will be latched into the AUX register.

Decode cycle: register source

MOV A, R5

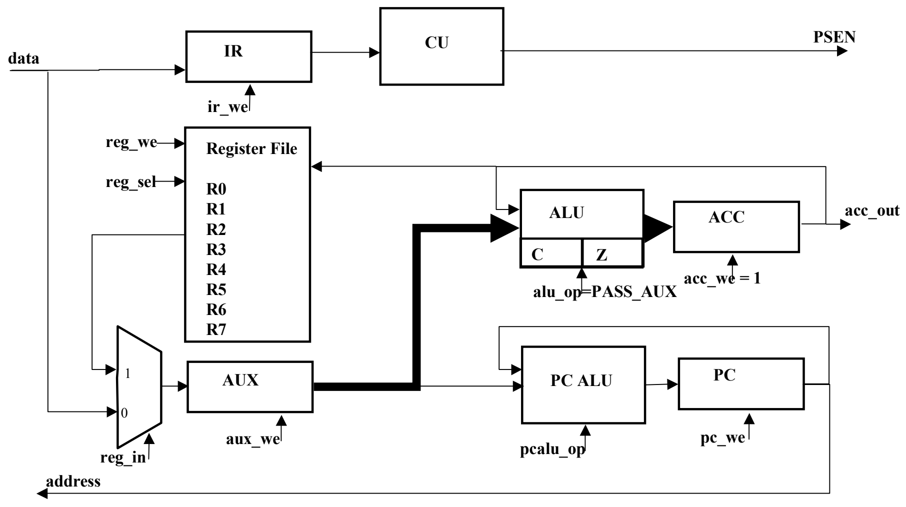


Here is an instruction which reads a value from a register: MOV A, R5. In this case, IR would contain B"1110_1101" or 0xED. The lower three bits of the instruction register are used as the reg_sel input to the register file, to select register R5 in this case. The input of the AUX multiplexor is set to 1 to cause the register file's output to go to the AUX register. On the next rising clock edge, the value of R5 will be latched into the AUX register.

For register and immediate source instructions, the AUX register will contain some value at the end of the decode cycle. What gets done with this value depends on the current opcode. Let's look at the execution of this particular instruction.

Execute cycle:

MOV A, R5



Recall that for MOV A, R5 the lower three bits of the IR are used to fetch the contents of register R5 from the register file into the AUX register during the decode cycle. During the execute cycle, the operand which was fetched is now available on the AUX output bus. All that is needed is for the ALU to be set up to pass the AUX input through to the ACC. At the end of the execute cycle the next rising edge clocks the new value into ACC.

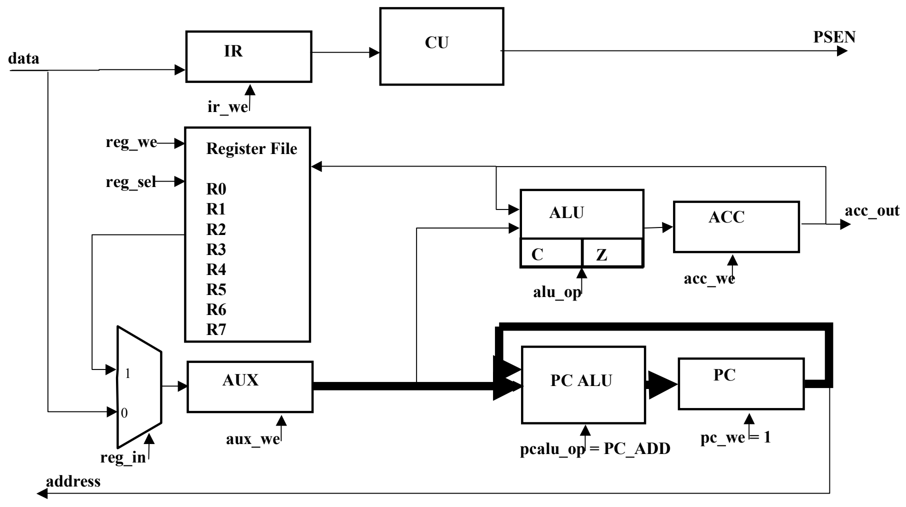
All other arithmetic instructions are similar. An ADDC instruction would simply set up the ALU for addition and at the end of the cycle, ACC would be loaded with the sum of AUX, ACC, and C.

By now you may have noticed that the registered ALU is similar to the one you may have designed in Comp Eng 112. The PC and its associated adder and incrementer are a simpler version of registered ALU. The AUX register and its multiplexor are even simpler. All these make up what is normally called the *data path* portion of Wimp51. The control unit and associated instruction decoder is what makes Wimp51 more interesting than a collection of logic and flip flops.

Lets finish up the execute cycle with a discussion of the operation during a SJMP instruction.

Execute cycle:

SJMP loop



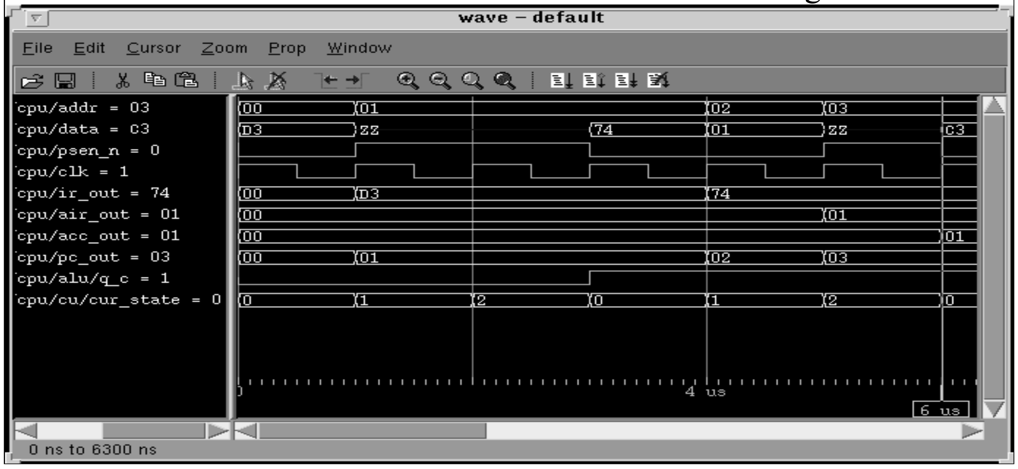
For a jump instruction, the value of the AUX register is added to the value of the PC. On the next rising clock edge, the PC will be loaded with this new value. Note that since jump instructions (SJMP and JZ) are two bytes long, the PC will have been incremented twice before the value of AUX is added to the PC. This means that you must calculate the relative offset in relation to the instruction immediately AFTER the jump instruction. 02h + 0FEh = 00h so the instruction "SJMP 0FEh" will continually jump to the same instruction!

Wimp51 is now ready to begin another fetch cycle to fetch the instruction stored at location *loop*.

That's about all there is to it. Bigger processors just increase the sizes of everything, do more things in parallel, sometimes attempt to work on more than one instruction at a time, and other tricks to speed things up. A few go the other direction. For example, Wimp51 is an eight bit processor and instructions are 8 or 16 bits wide. It would be useful for everything to be a uniform eight bits. So called four bit processors (you are probably wearing one on your arm now) use more than one four bit nibble to make up an instruction. This is also typical of larger processors including the ubiquitous pentium. Instructions are made up of one or more bytes and are read in chunks of four or more bytes at a time. There are enough complications to warrant three or more additional courses on this topic of *computer organization* but this is enough for now!

Wimp51 timing diagram

- •Two complete instruction cycles are shown
- •First is a SETB C (D3h) at location 0000
- •PC increments at end of Fetch cycle, C set at end of Exec cycle
- •What is second instruction? Value of the data bus during fetch #2?

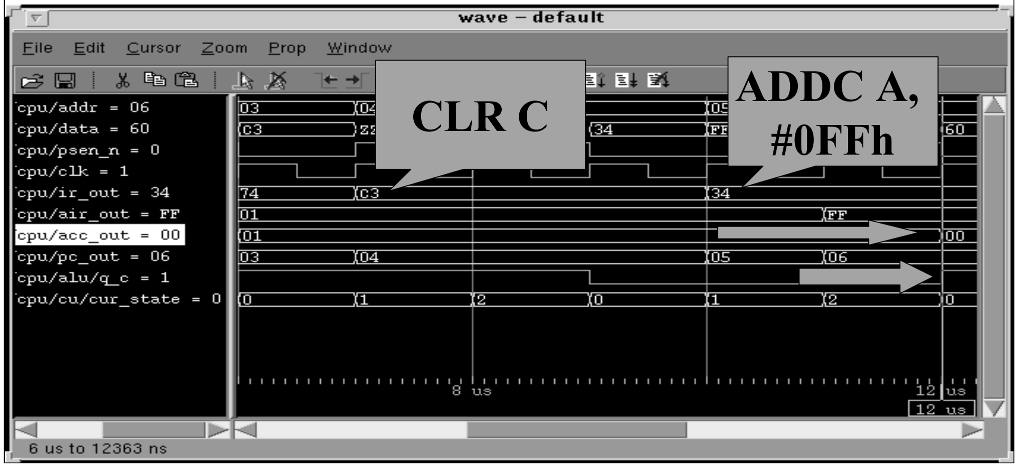


```
cpu/addr = 03           00 | 01 | 02 | 03
cpu/data = C3           D3 | ZZ | 74 | 01 | ZZ | C3
cpu/psen_n = 0
cpu/clk = 1
cpu/ir_out = 74         00 | D3 | 74
cpu/air_out = 01        00 | 01
cpu/acc_out = 01        00 | 01
cpu/pc_out = 03         00 | 01 | 02 | 03
cpu/alu/q_c = 1
cpu/cu/cur_state = 0    0 | 1 | 2 | 0 | 1 | 2 | 0
```

0 ns to 6300 ns

Wimp51 timing diagram 2

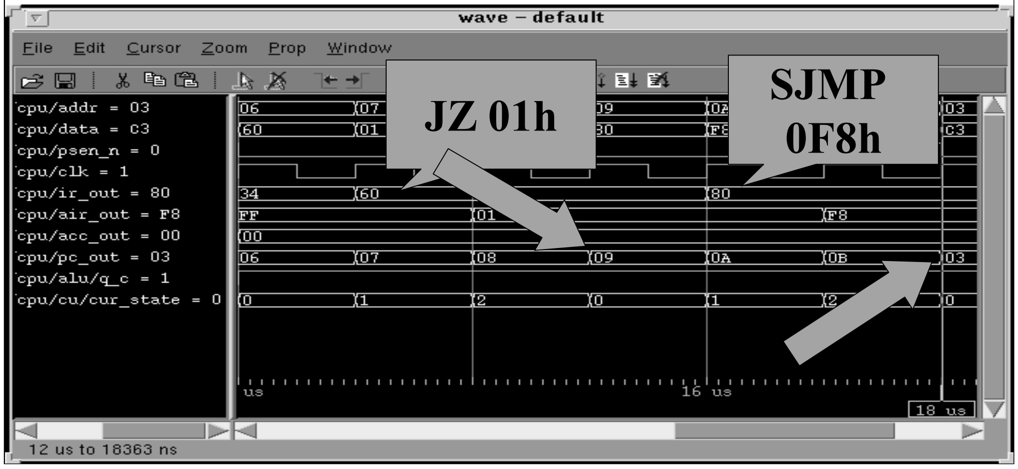
- Two more instructions. What are they?
- Note ACC goes from 1 to 0 at end of #2
- C is set at end of instruction #2



```
cpu/addr = 06
cpu/data = 60
cpu/psen_n = 0
cpu/clk = 1
cpu/ir_out = 34
cpu/air_out = FF
cpu/acc_out = 00
cpu/pc_out = 06
cpu/alu/q_c = 1
cpu/cu/cur_state = 0
```

Wimp51 timing diagram 3

- Two more instructions. What are they?
- First instruction increments PC once more
- Instruction #2 increments PC then changes it to 3h



```
cpu/addr = 03
cpu/data = C3
cpu/psen_n = 0
cpu/clk = 1
cpu/ir_out = 80
cpu/air_out = F8
cpu/acc_out = 00
cpu/pc_out = 03
cpu/alu/q_c = 1
cpu/cu/cur_state = 0
```

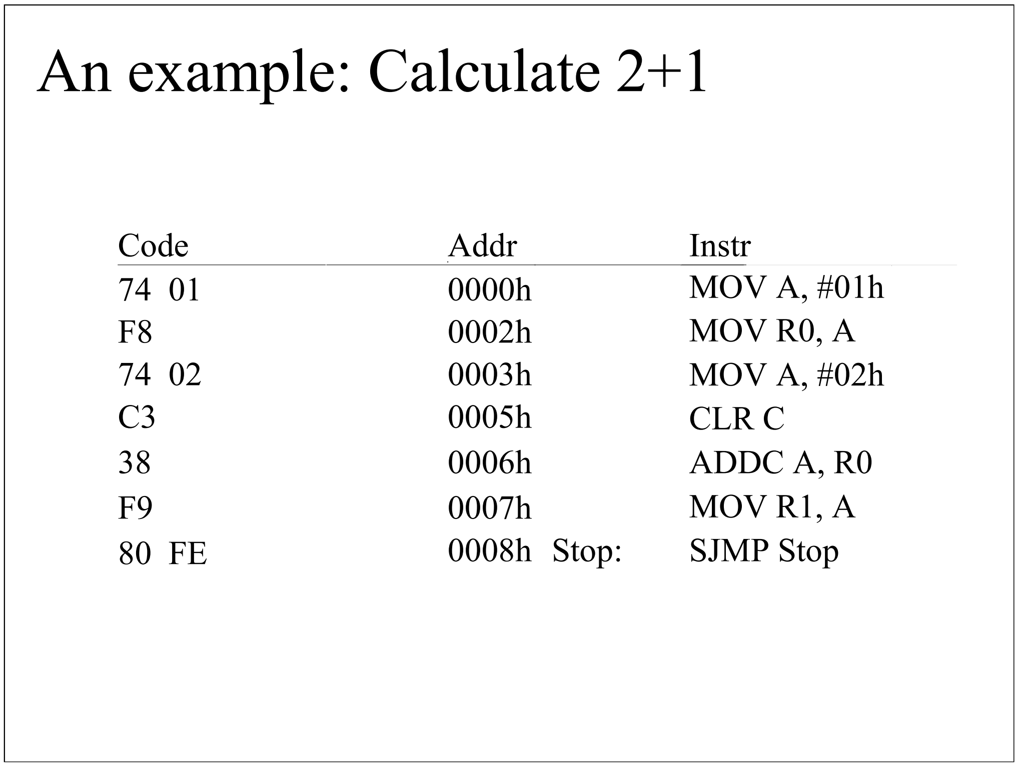
12 us to 18363 ns

Wimp51 timing

- Registers get loaded by rising edge at end of cycle
- Each instruction lasts three clock cycles
- Each instruction cycle consists of a Fetch, Decode, and Execute clock cycle
- True 8051 is similar but more options so more complex (12 or more clock cycles per instruction)

Wimp51 timing

- Timing diagrams generated by a ***simulator***
- Simulator 'executes' a ***model*** of Wimp51
- Model is written in ***VHDL***
- VHDL can be used to ***synthesize*** Wimp51 hardware
- Simulation models let us 'try out' a design before committing to hardware
- We can simulate hardware, software, or both
- Used for ***rapid prototyping*** and to ***get it right the first time***



An example: Calculate 2+1

Code	Addr		Instr
74 01	0000h		MOV A, #01h
F8	0002h		MOV R0, A
74 02	0003h		MOV A, #02h
C3	0005h		CLR C
38	0006h		ADDC A, R0
F9	0007h		MOV R1, A
80 FE	0008h	Stop:	SJMP Stop

This is a simple program to add 1 plus 2. Register R0 is loaded with a '1', then the accumulator is loaded with a 2, added to register 0 and the result stored in register 1. We would express this in C as sum=1+2 where sum is assigned to location 1. Note the use of a 'branch to self' to bring the program to a halt since the Wimp51 has no halt instruction.

Assignment:

1. Write down the machine code corresponding to these instructions.
2. What determines the value of 'stop' and thus the code for the SJMP instruction?
3. How long does it take to execute this program in units of clock cycles?

An example: Calculate 2+1

PC	R0	R1	Acc	C		Instruction
0	??	??	??	?		MOV A, #1
2	??	??	1	?		MOV R0, A
3	1	??	1	?		MOV A, #2
5	1	??	2	?		CLR C
6	1	??	2	0		ADDC A, R0
7	1	??	3	0		MOV R1, A
8	1	3	3	0	Stop:	SJMP Stop

This is a simple program to add 1 plus 2. Register R0 is loaded with a '1', then the accumulator is loaded with a 2, added to register 0 and the result stored in register 1. We would express this in C as sum=1+2 where sum is assigned to location 1. Note the use of a 'branch to self' to bring the program to a halt since the Wimp51 has no halt instruction.

Assignment:

1. Write down the machine code corresponding to these instructions.
2. What determines the value of 'stop' and thus the code for the jmp instruction?
3. How long does it take to execute this program in units of clock cycles?

Calculate 3*5 with for loop

Code	Addr	Instr
		MOV A, #0
		MOV R0, A
		MOV A, #3
		MOV R1, A
	Loop:	MOV A, #5
		CLR C
		ADDC A, R0
		MOV R0, A
		MOV A, R1
		CLR C
		ADDC A, #0FFh
		MOV R1, A
		JZ Stop
		SJMP Loop
	Stop:	SJMP Stop

Here is a simple program which implements a for loop to multiply 5 times 3 by adding 5 to itself 3 times. Data location 0 is used to hold the product and location 1 is used to hold the loop counter. Note the addition of –1 (FF in two's complement notation) since the Wimp51 has no subtract instruction. Also note the use of the jump if zero instruction to implement the end of loop test. This is known as a conditional branch.

Assignment:

1. Write down the machine code for this program.
2. How long does it take to execute this program in clock cycles?
3. Discuss how you would modify Wimp51 to:
   a) implement bit addressable ram
   b) simple parallel port (8-bit) I/O.
   c) a subroutine call.

4. List all the registers that are used by this program and list their contents after each instruction is executed. Do this until the instruction at 'stop' is executed. Does this program in fact multiply 3*5?
5. How would you modify this program to multiply 21*60? Keep in mind that the result will be larger than eight bits!