

We are going to show from beginning to end how to wire up a generic 8-bit machine. This machine will use a 2-operand format, meaning that instructions are of the time  $A=A+B$ . So, for example, "Add r0, r1" is  $r0=r0+r1$ .

The machine is byte-addressable. Offsets are sign-extended, and jumps are done by adding the sign-extended offset field to the PC. Immediates are not sign-extended.

The machine has 3 different instruction formats: A, B, and C.

A-type:	Opcode	ds	s	extra
	7-4	3	2	1-0

B-type:	Opcode	ds	Immediate	
	7-4	3	2-0	

C-type:	Opcode	Offset	
	7-4	3-0	

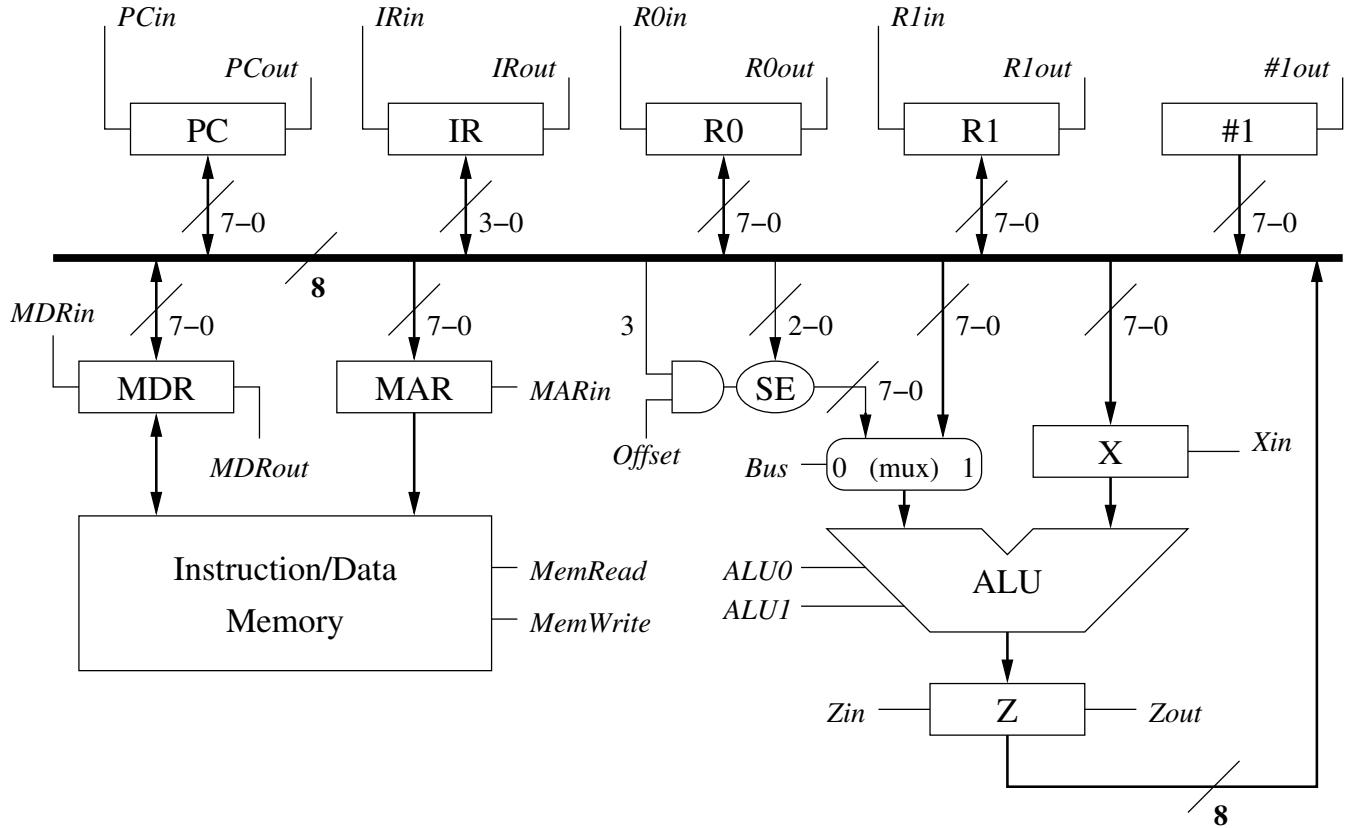
The ALU can perform 4 functions:

<b>Operation</b>	<b>ALU0</b>	<b>ALU1</b>
Add	1	1
Or	1	0
And	0	1
Not A	0	0

Here are a few of the instructions that have been defined:

<b>Name</b>	<b>Opcode</b>	<b>Operation</b>
add	0001	$rds=rds+rs$
addm	0010	$rds=rds+mem[rs]$
addi	0011	$rds=rds+imm$
jmp	1111	PC=PC+offset (offset is sign extended)

On the following page is a diagram of the machine. The control signals are in italics. MAR does not need an "out" signal, because the MemRead and MemWrite signals control whether or not memory looks at the MAR signal. The X register does not need an "out" signal either - the ALU will always perform an operation, which we can ignore if we want. The "#1" register contains the number 1, which is used to increment the PC. The SE block is sign-extend logic, and needs some further explanation. The problem is that the ALU requires an 8-bit value, but the Immediate field is only 3 bits wide, and the offset field is only 4 bits wide and also needs to be sign-extended. The sign extend logic creates a 5-bit value which matches the contents of bit 3, so that the 8-bit value generated looks like 33333210 (instead of 76543210). The AND gate is necessary because an immediate instruction is not sign-extended, and there needs to be a way to ensure that the top 5 bits are all set to zero.



Here are the 21 control signals.

PCin	PCout	IRin	IRout	R0in	R0out	#1out
R1in	R1out	MDRin	MDRout	Zin	Zout	MARin
Xin	ALU0	ALU1	MemRead	MemWrite	Bus	Offset

In order to execute an instruction, we must make sure that each of these control signals is set to the appropriate value at the appropriate time. For example, here are the steps necessary to fetch an instruction from memory:

S t e p	P C o u t	P C o u t	I R i n u t	I R i n u t	R 0 o u t	R 0 o u t	R 1 o u t	R 1 o u t	M D R i n	M D R i n	Z i n	Z o u t	M A R i n	X i n	# 1 o u t	A L U 0	A L U 1	M r e a d	M w r i t e	B u s	O f f s e t
<b>0</b>	0	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	1	0	0
<b>1</b>	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	1	1	1	0	1
<b>2</b>	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0
<b>3</b>	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0

Once the instruction has been fetched, it must be executed. Here is how  $R0 = R0 + R1$  would be performed:

S t e p	P C i n	P C o u	I R i n	I R o u	R 0 i n	R 1 o u	R 1 o u	M D R i n	M D R o u	Z i n	Z o u	M A R i n	X i n	# 1 o u	A L U 0	A L U 1	M r e a d	M w r i t e	M w r i t e	B u s	O f f s e t
0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	1	1	0	0	1	0
2	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0

Here is how a Jump would be performed (0's are removed for legibility reasons):

S t e p	P C i n	P C o u	I R i n	I R o u	R 0 i n	R 1 o u	R 1 o u	M D R i n	M D R o u	Z i n	Z o u	M A R i n	X i n	# 1 o u	A L U 0	A L U 1	M r e a d	M w r i t e	M w r i t e	B u s	O f f s e t
0		1												1							
1			1							1						1	1				1
2	1										1										

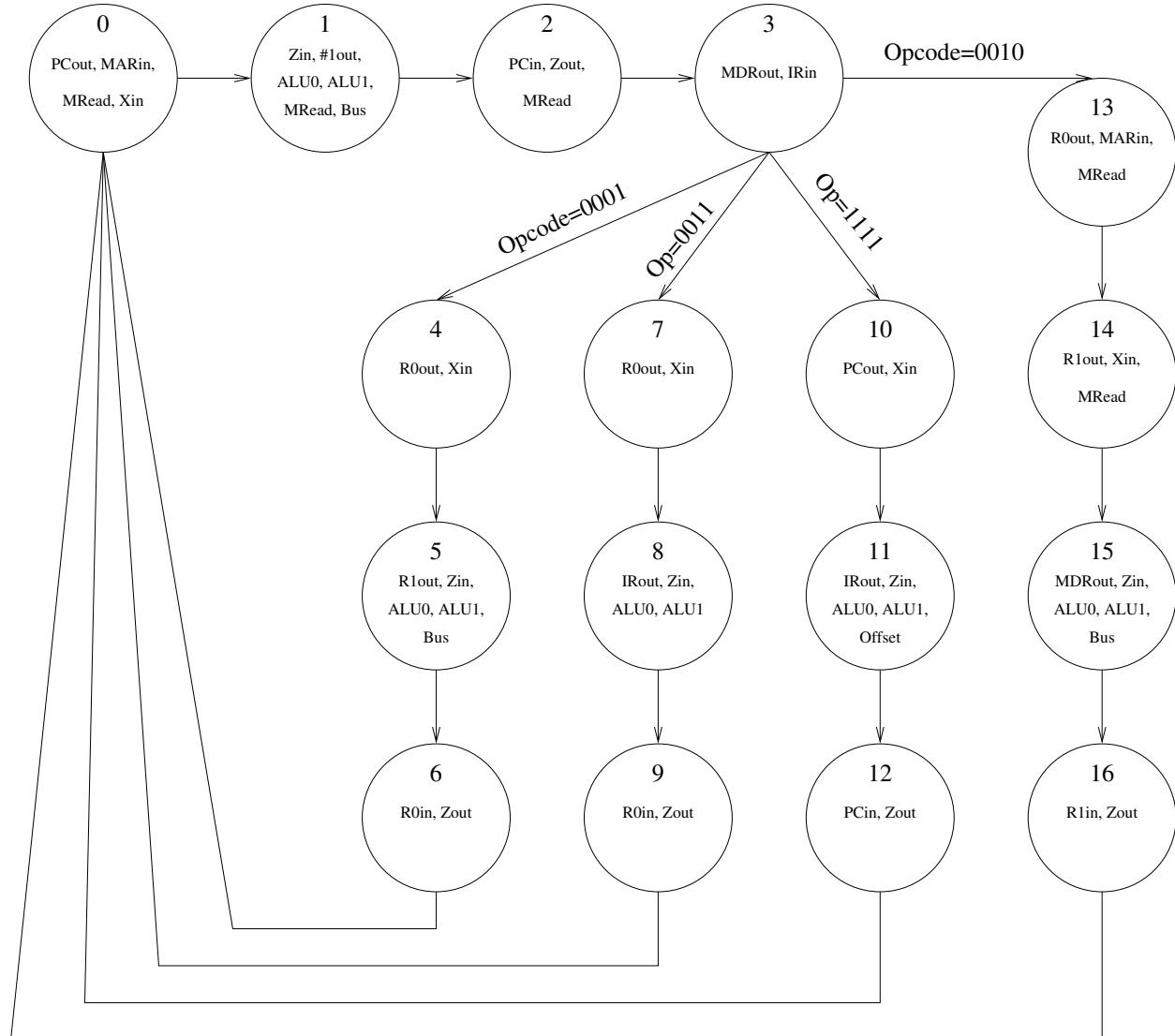
Here is an Immediate ( $R0 = R0 + immd$ ) instruction:

S t e p	P C i n	P C o u	I R i n	I R o u	R 0 i n	R 1 o u	R 1 o u	M D R i n	M D R o u	Z i n	Z o u	M A R i n	X i n	# 1 o u	A L U 0	A L U 1	M r e a d	M w r i t e	M w r i t e	B u s	O f f s e t
0						1									1						
1				1							1						1	1			
2					1							1									

A memory access ( $R1 = R1 + \text{mem}[R0]$ ):

S t e p	P C i n	P C o u	I R i n	I R o u	R 0 i n	R 1 o u	R 1 o u	M D R i n	M D R o u	Z i n	Z o u	M A R i n	X i n	# 1 o u	A L U 0	A L U 1	M r e a d	M w r i t e	M w r i t e	B u s	O f f s e t
0						1									1				1		
1							1								1				1		
2										1	1						1	1			1
3							1					1									

Once every instruction has been fully specified in this manner, one can create sequential logic circuits which will cause each control signal to be set to the right value at the right time. In the following diagram, the State number is in the top center of each circle, and inside the circle are the signals that need to be asserted that cycle.



From this diagram we can create the following table, which describes the exact boolean equation for every control signal. Since there are 17 states, we will need 5 state variables, Y4-Y0. In this table, "State0" = 00000, "State1" = 00001, etc. Remember, when doing sequential circuits, both the current outputs and the next state values must be calculated. In this table, the NextState values are calculated based on the current state and the current inputs. (The current inputs are I7-I4, which are the opcode bits and come from the Instruction Register). This is a Moore model, since the outputs are a function of the current state only.

<b>PCin=</b>	State2 + State12	<b>PCout=</b>	State0 + State10
<b>IRin=</b>	State3	<b>IRout=</b>	State8 + State11
<b>R0in=</b>	State6 + State9	<b>R0out=</b>	State4 + State7 + State13
<b>R1in=</b>	State16	<b>R1out=</b>	State5 + State14
<b>MDRin=</b>		<b>MD Rout=</b>	State3 + State15
<b>Zin=</b>	State1 + State5 + State8 + State11 + State15		
<b>Zout=</b>	State2 + State6 + State9 + State12 + State16		
<b>MARin=</b>	State0 + State13	<b>#1out=</b>	State1
<b>Xin=</b>	State0 + State4 + State7 + State10 + State14		
<b>ALU0=</b>	State1 + State5 + State8 + State11 + State15		
<b>ALU1=</b>	State1 + State5 + State8 + State11 + State15		
<b>Mread=</b>	State0 + State1 + State2 + State13 + State14	<b>Mwrite=</b>	
<b>Bus=</b>	State1 + State5 + State15	<b>Offset</b>	State11
<b>NextState0=</b>	State6 + State9 + State12 + State16	<b>NextState1=</b>	State0
<b>NextState2=</b>	State1	<b>NextState3=</b>	State2
<b>NextState4=</b>	State3 * $\overline{I7} * \overline{I6} * \overline{I5} * I4$	<b>NextState5=</b>	State4
<b>NextState6=</b>	State5	<b>NextState7=</b>	State3 * $\overline{I7} * \overline{I6} * I5 * I4$
<b>NextState8=</b>	State7	<b>NextState9=</b>	State8
<b>NextState10=</b>	State3 * $I7 * I6 * I5 * I4$	<b>NextState11=</b>	State10
<b>NextState12=</b>	State11	<b>NextState13=</b>	State3 * $\overline{I7} * \overline{I6} * I5 * \overline{I4}$
<b>NextState14=</b>	State13	<b>NextState15=</b>	State14
<b>NextState16=</b>	State15		

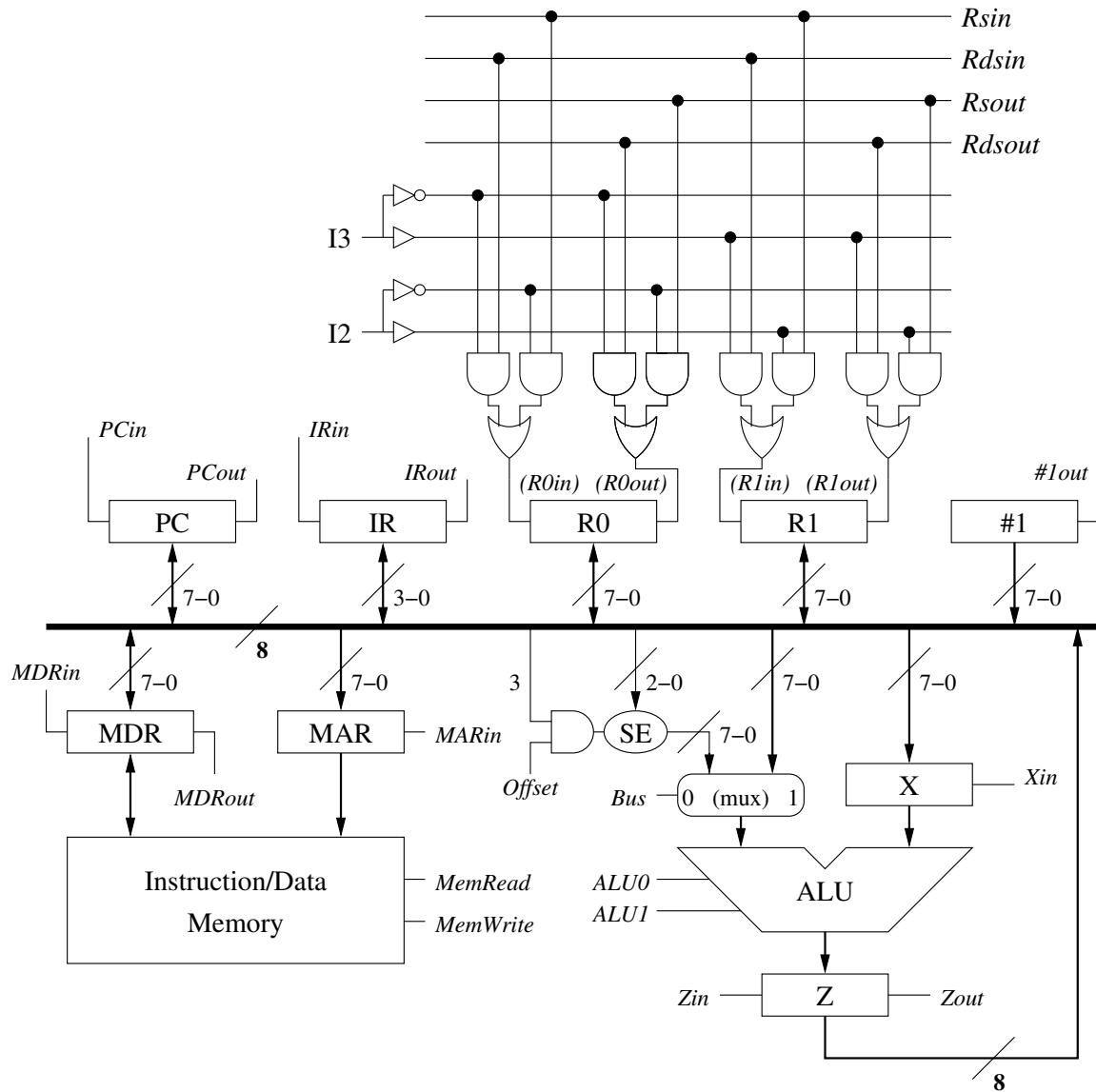
Here, for example, are the exact boolean equations for a couple of signals:

$$PCin = (\overline{Y4} * \overline{Y3} * \overline{Y2} * Y1 * \overline{Y0}) + (\overline{Y4} * Y3 * Y2 * \overline{Y1} * \overline{Y0})$$

$$R1in = (Y4 * \overline{Y3} * \overline{Y2} * \overline{Y1} * \overline{Y0})$$

$$NextState4 = \overline{Y4} * \overline{Y3} * \overline{Y2} * Y1 * Y0 * \overline{I7} * * \overline{I6} * \overline{I5} * I4$$

As I discussed in class, the actual diagram needs to be modified a bit in order to incorporate the fact that the instruction specifies which register is to be the source and which is to be the destination. The modified diagram would look like this:



This would not require any new signals, we would just need to rename 4 of them. *R0in*, *R0out*, *R1in*, and *R1out* would become *Rdsin*, *Rdsout*, *Rsin*, and *Rsout*.

Here are the 21 control signals.

PCin	PCout	IRin	IRout	Rdsin	Rdsout	#1out
Rsin	Rsout	MDRin	MDRout	Zin	Zout	MARin
Xin	ALU0	ALU1	MemRead	MemWrite	Bus	Offset

The microcode steps in order to perform an add would look like this:

S t e p	P C o u t	P C o u t	I R i n o u t	I R i n o u t	R d s o u t	R d s o u t	R s o u t	M D R i n	M D R i n	Z i n	Z o u t	M A R i n	X i n	# 1 o u t	A L U 0	A L U 1	M r e a d	M w r i t e	B u s	O f f s e t	
0															1						
1											1				1		1	1			1
2					1								1								

The machine looks like this:

