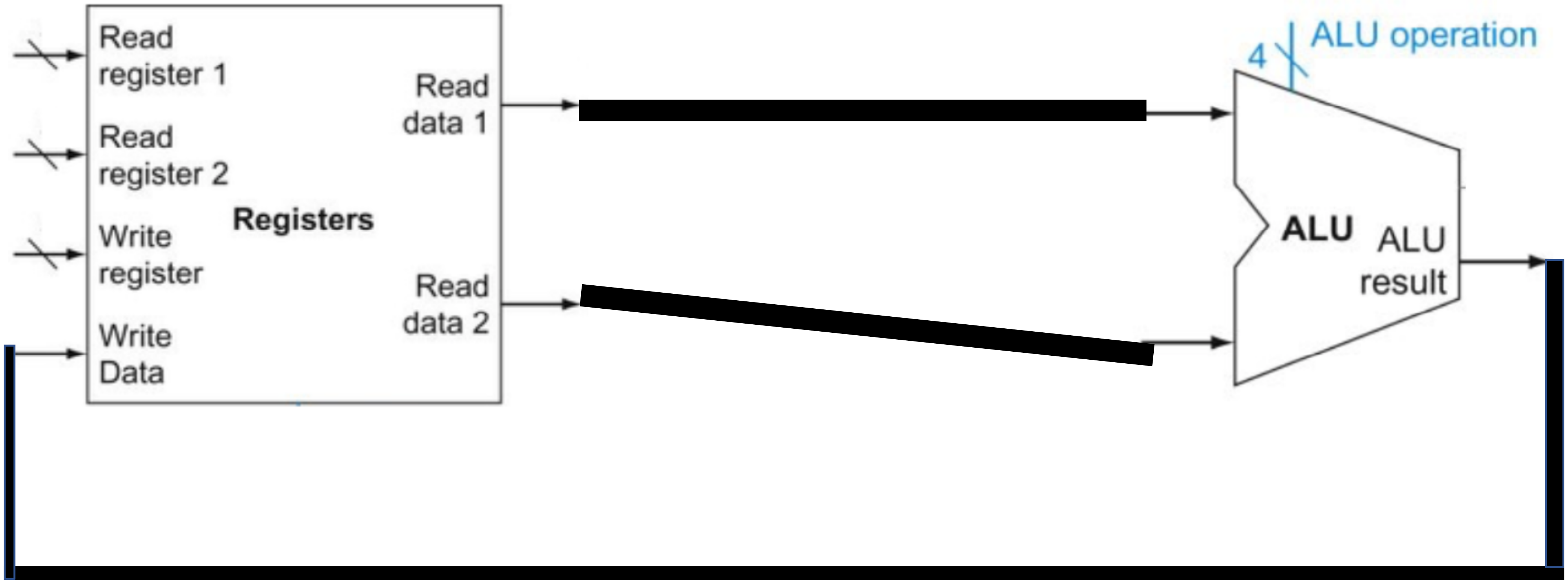
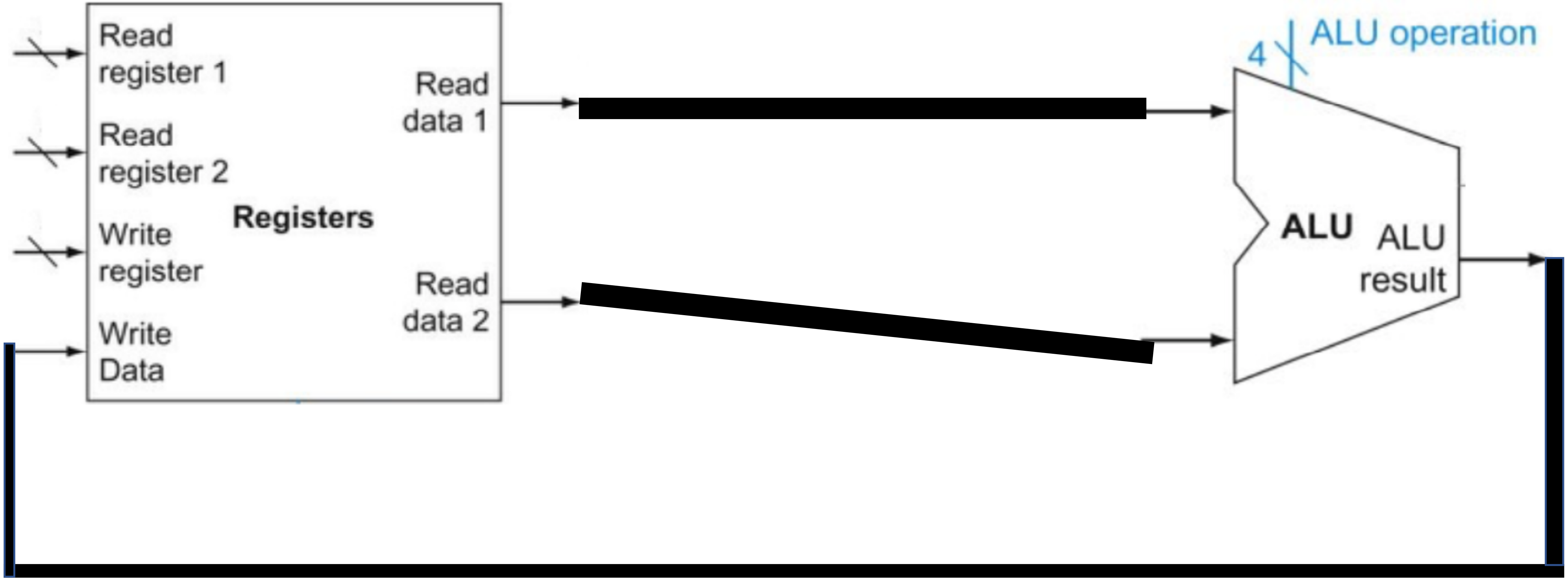
Automatic Computer

Nathan Bowman


Read register 1
Read register 2
Write register
Write Data
Registers
Read data 1
Read data 2
4
ALU operation
ALU
ALU result



ReadReg1	ReadReg2	WriteReg	Op

Combining inputs

Simple machine had 4 registers, so 2 bits necessary to specify register

Total bits:

* first read register – 2
* second read register – 2
* write register – 2
* operation – 4

10 bits total

Combining inputs

Rather than considering as four separate inputs, think of as single 10-bit input that controls everything

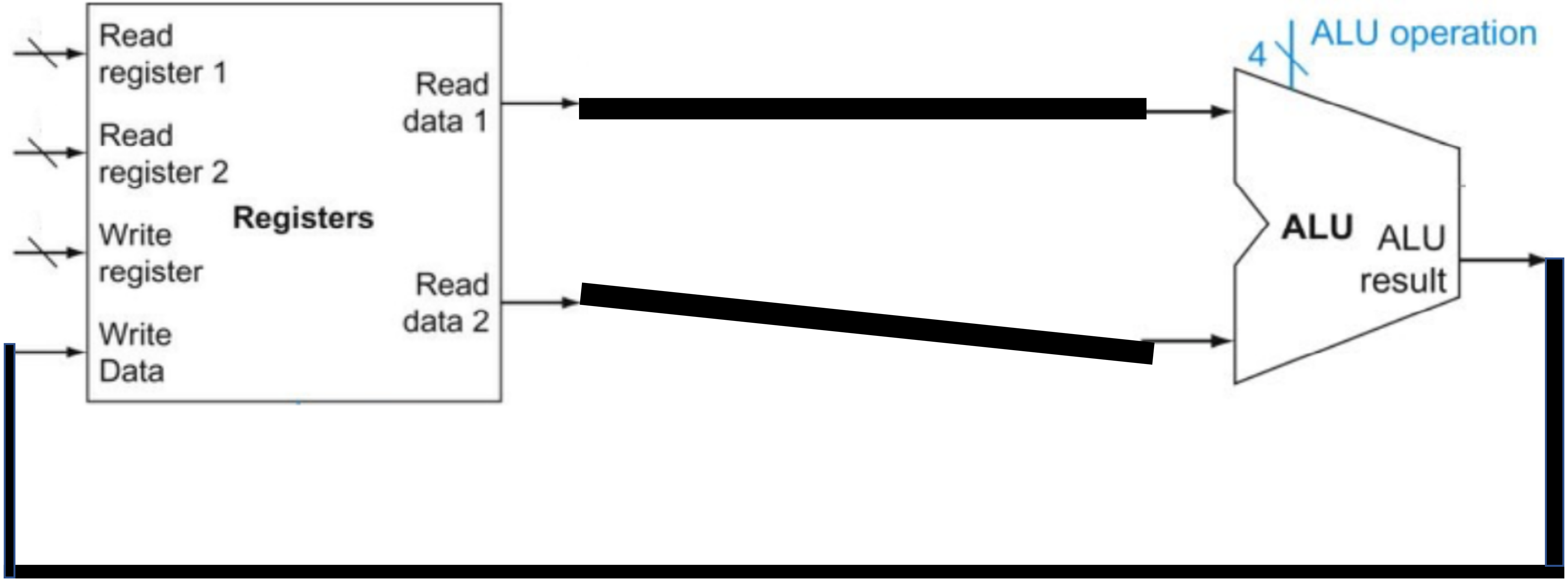
This changes *nothing* about how circuit operates – just split 10-bit input back to inputs we had before

Only thing we need to be careful of is splitting bits into correct groups

Combining Inputs

Assume bits numbered starting with least-significant bit

9-8	7-6	5-4	3-0
ReadReg1	ReadReg2	WriteReg	Op


Read register 1
Read register 2
Write register
Write Data
Registers
Read data 1
Read data 2
4
ALU operation
ALU
ALU result

Moving Through Inputs

Each instruction is now just 10-bit input to circuit

To make computer run automatically, we need to

* save list of instructions somewhere
* move through list of instructions as clock ticks

Memory

For purposes of course, memory will be considered black box

All we need to know about memory is its input/output specification:

- * Input: address of byte
- * Output: value of byte

Simple lookup table

Using Memory for Instructions

Memory stores bytes, which we typically think of as numbers

As we’ve seen, *an instruction is just a number*

This is key insight to making computers work: instructions are just another form of data that can be stored

Cheating a little

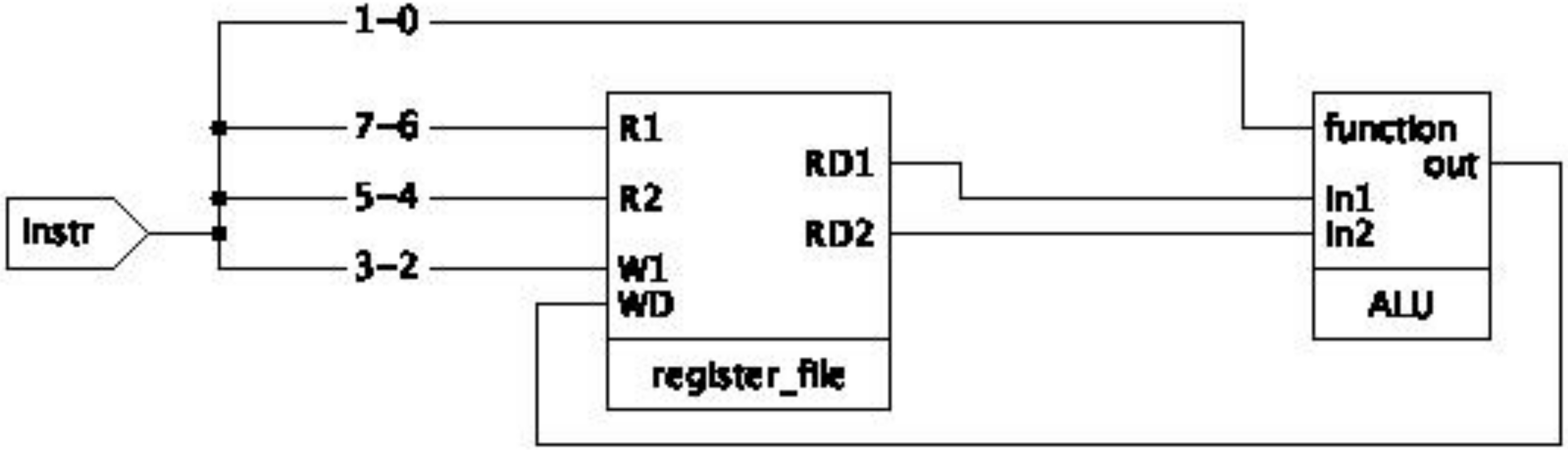
Memory grabs byte (8 bits) at a time, but our instructions were 10 bits

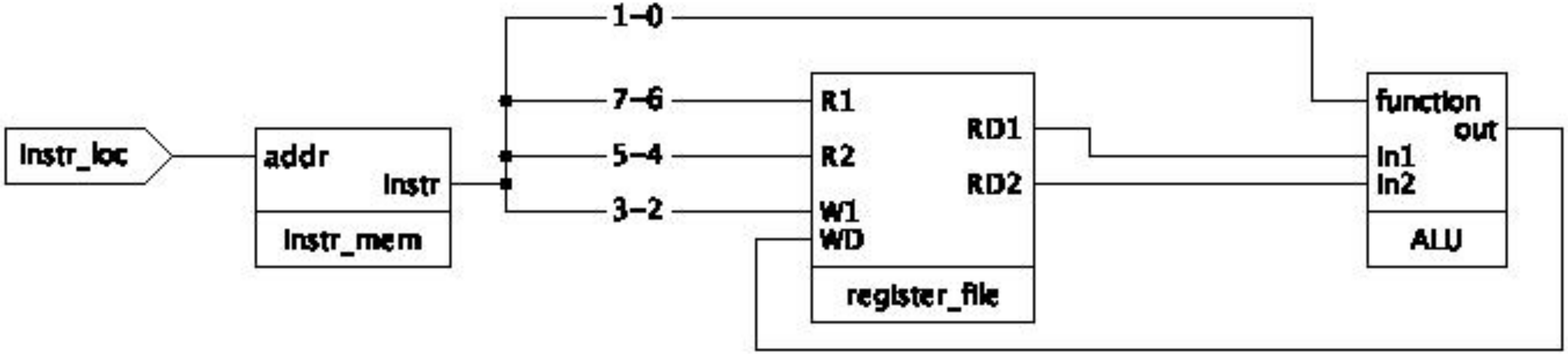
For this reason (among others), actual architectures tend to have instruction lengths that are multiples of 8

For simplicity, we’ll pretend our ALU only has 4 operations

7-6	5-4	3-2	1-0
ReadReg1	ReadReg2	WriteReg	Op

One instruction = one byte = one thing we can grab from memory


1–0
7–6
5–4
3–2
Instr
R1
R2
W1
WD
RD1
RD2
register_file
function
out
In1
In2
ALU


Instr_loc
addr
Instr
Instr_mem
1–0
7–6
5–4
3–2
R1
R2
W1
WD
RD1
RD2
register_file
function
out
In1
In2
ALU

Memory addresses

We can now pre-load our instructions into memory and choose which instruction we want

One caveat: how many bits does instr_loc need?

Same question another way: how many bits needed to specify memory address?

Memory addresses

Answer: depends on the size of memory!

Real MIPS memory is 2^{32} bytes (we’re going to see 32 come up a lot with MIPS)

We have no great reason to choose any particular number yet, so we will stick with memory as 2^{32} bytes

To choose between 2^{32} things, we need…

Memory addresses

So, assume instr_loc is 32-bit input that specifies *location* of instruction in memory

Way back when, we said the point of this was to make computer automatic. How does this help?

If we are at instruction 1, we want to go to instruction 2. From 2, we go to 3…

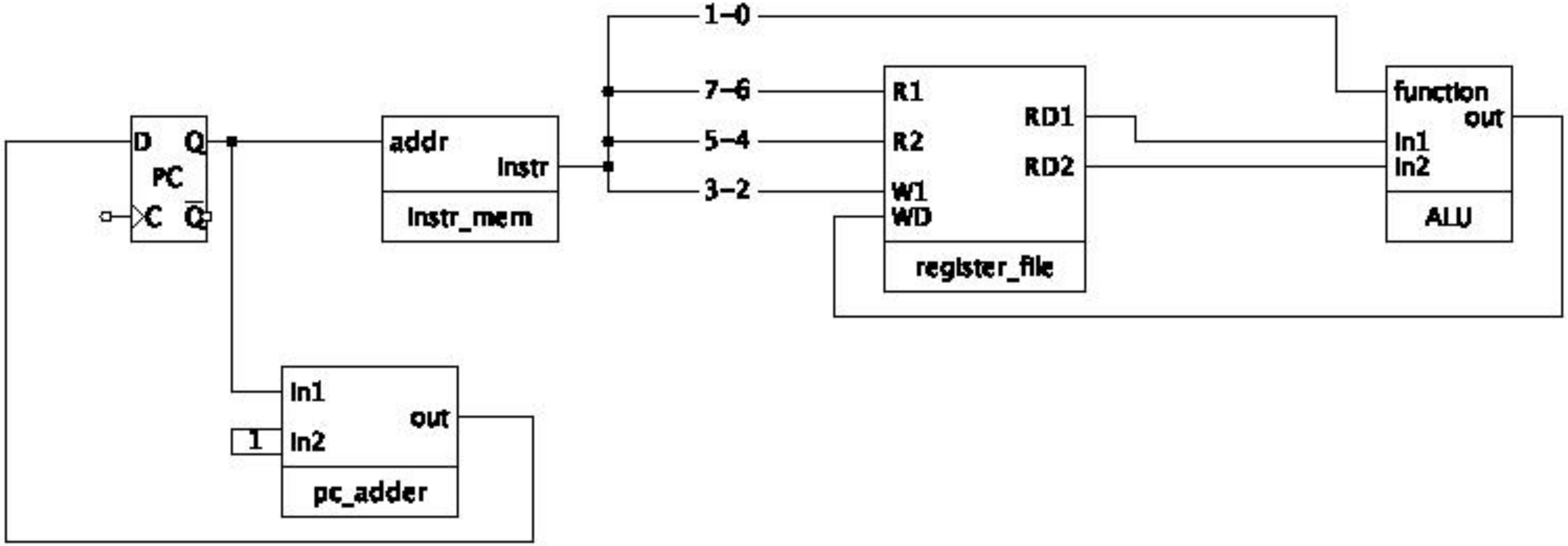
Automatic Computer

Two simple additions to diagram

* register to store current instruction *location* (after all, it is state)

* adder to increment register

Since register is tied to clock, machine will do 1 instruction per clock cycle, then move to next instruction!


D
Q
PC
C
Q̄
addr
instr
instr_mem
1–0
7–6
5–4
3–2
R1
R2
W1
WD
RD1
RD2
register_file
function
out
in1
in2
ALU
in1
in2
1
out
pc_adder

Ta-da!

That’s it! New register called **program counter (PC)** maintains *location* of current instruction

PC attached to small adder that always adds fixed amount to move to next instruction – in this case, adds 1 because next instruction is 1 byte away

In 32-bit MIPS, instructions will be 4 bytes apart

And now you have a computer that can run a program