Implementing I-type Instructions

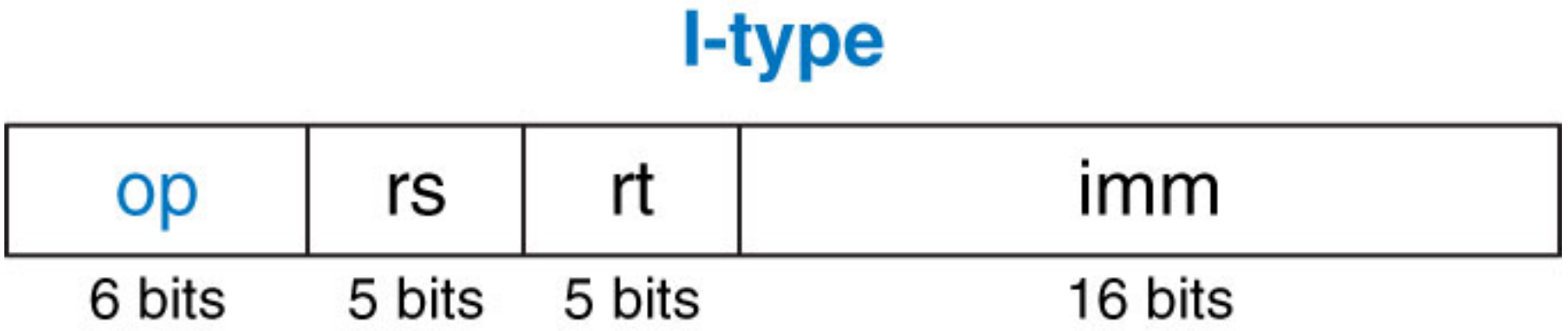
Author: Jared Moore

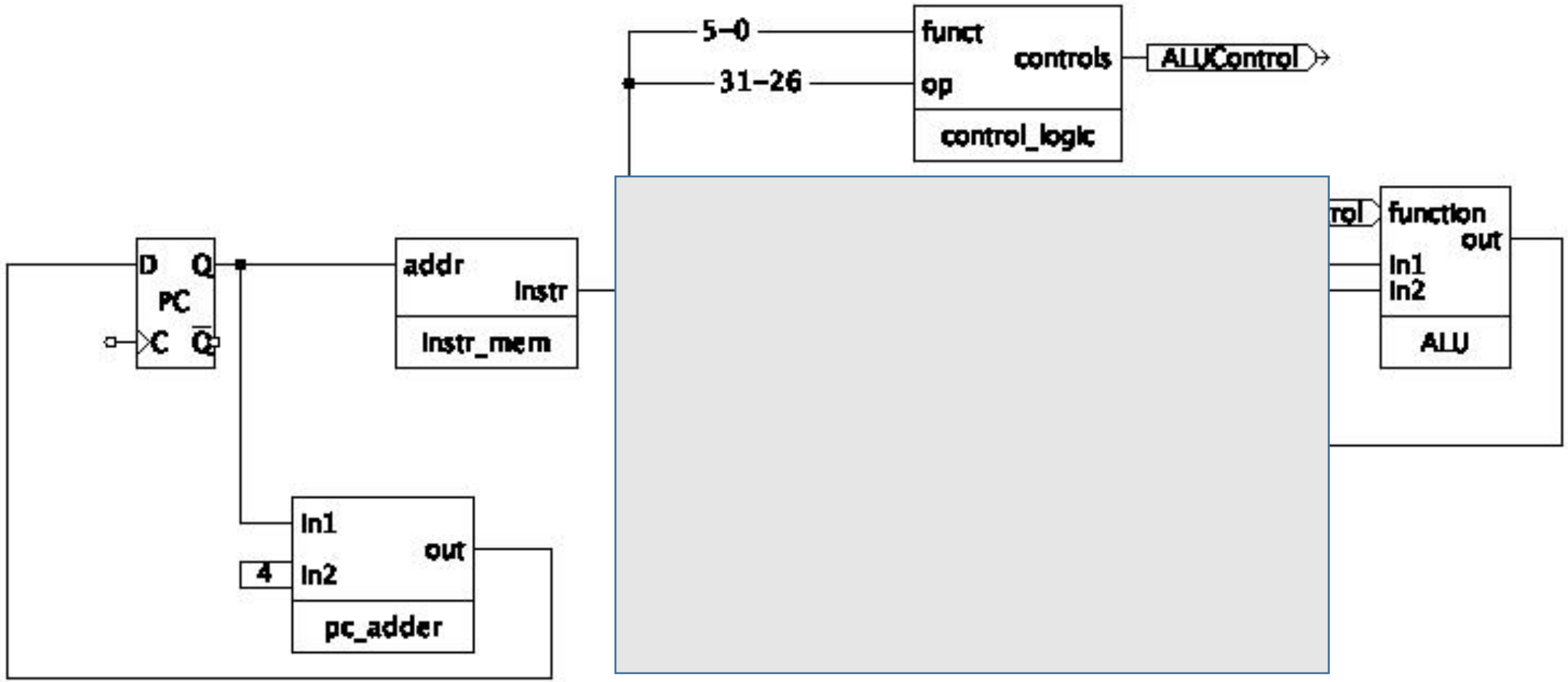
Edited by Nathan Bowman

I-Type Instructions

Implementing I-type instructions involves similar principles to what we have already seen

First, we design circuit for *just* I-type instructions




5–0
31–26
funct
op
controls
control_logic
ALUControl
D
Q
PC
C
Q
addr
Instr
Instr_mem
In1
In2
4
out
pc_adder
rol
function
out
In1
In2
ALU

I-Type

Program counter still proceeds in same manner

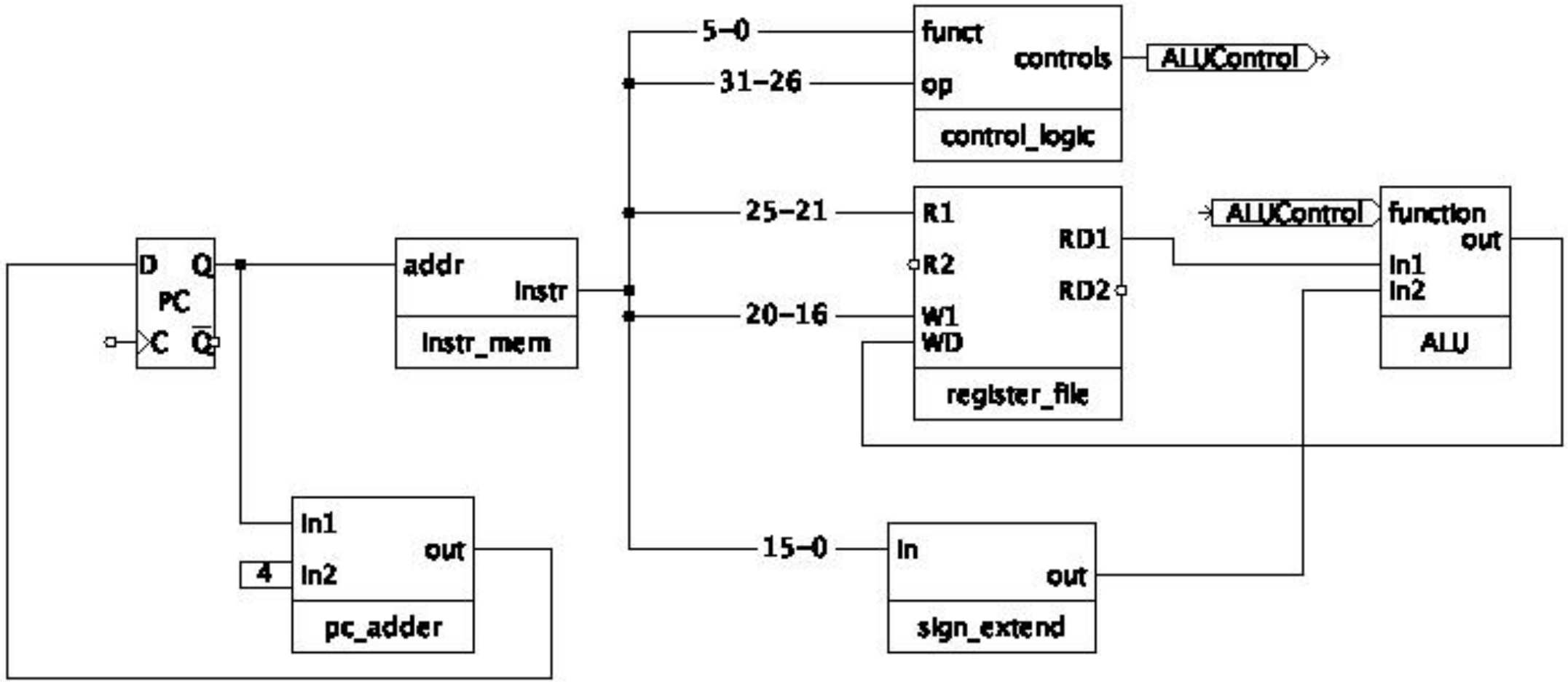
ALU still takes inputs and produces output that feeds back toward registers

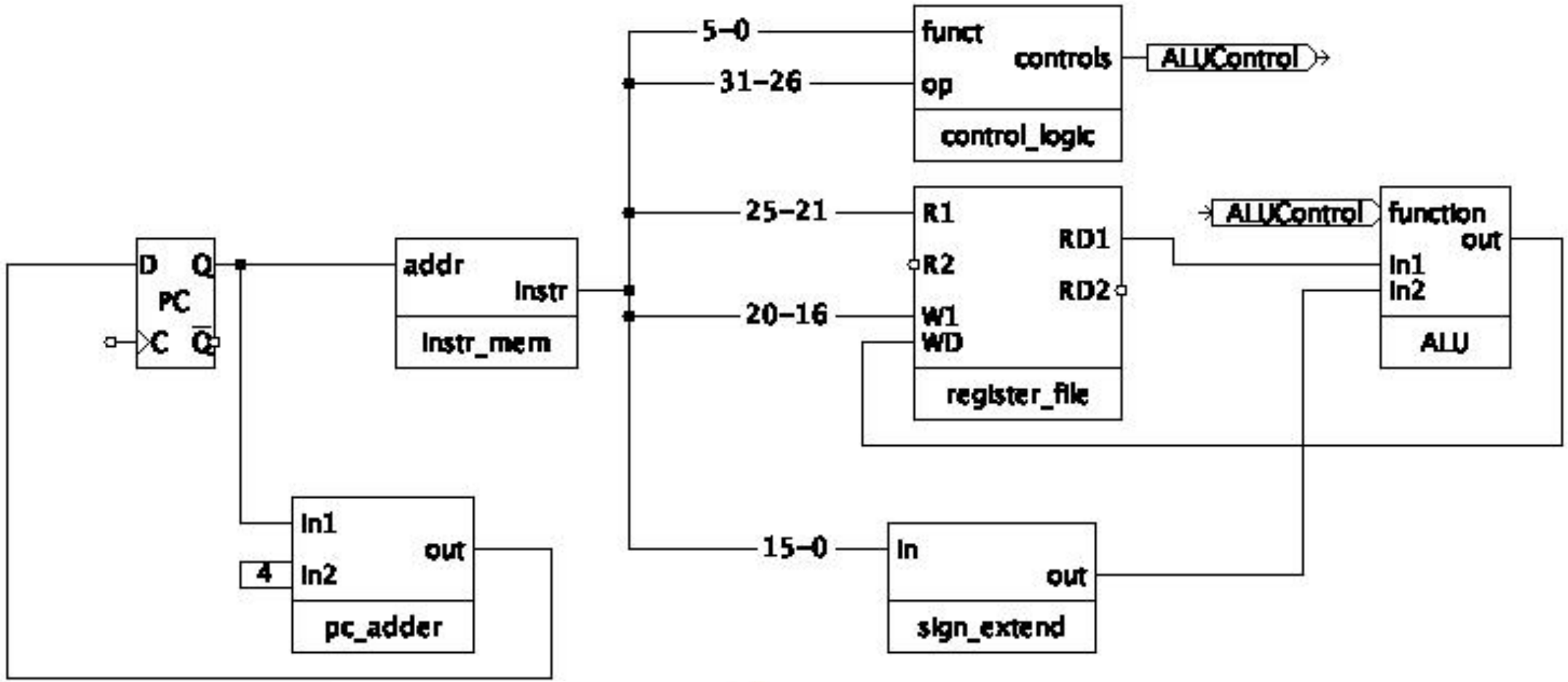
Control logic still looks at instruction to determine how ALU operates

- In this diagram, we still pass bits 0-5 to ALU, but it ignores them because I-type has no function code

I-Type

Main difference is how we **decode** instruction -- how we determine which bits go where after instruction read from memory


5–0
31–26
25–21
20–16
15–0
funct
op
controls
control_logic
ALUControl
R1
R2
W1
WD
RD1
RD2
register_file
ALUControl
function
In1
In2
out
ALU
D
Q
PC
C
Q
addr
Instr
Instr_mem
In1
4
In2
out
pc_adder
In
out
sign_extend



I-type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Simplifying assumption

Recall from previous lecture that not all immediates are sign-extended

Logical immediates are zero-extended

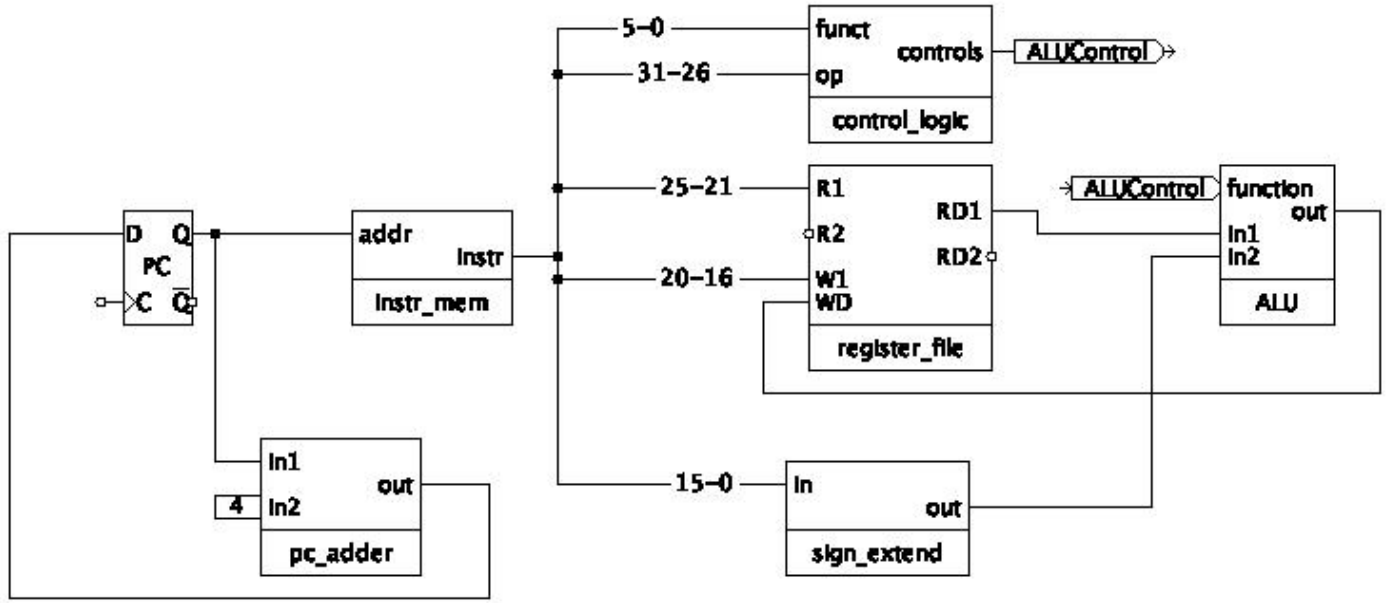
We could examine opcode and use mux to determine whether to use sign-extended or zero-extended output

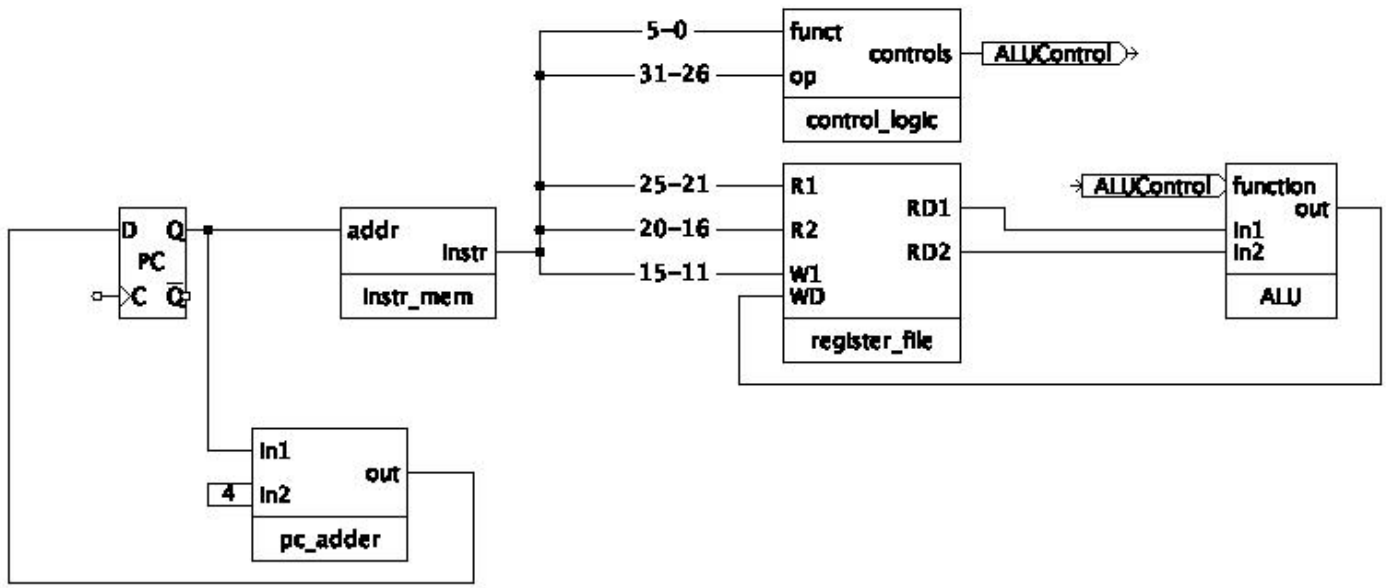
However, when designing our circuit, we will assume that all immediates are sign-extended (largely because your book makes this assumption)

I-type

Designing microarchitecture for I-type wasn't so bad

Fundamentally very similar to R-type (by design)


5–0
31–26
funct
op
controls
control_logic
ALUControl
25–21
20–16
R1
R2
W1
WD
RD1
RD2
register_file
ALUControl
function
In1
In2
out
ALU
D
Q
PC
C
Q
addr
Instr
Instr_mem
In1
4
In2
out
pc_adder
15–0
In
out
sign_extend


5–0
31–26
funct
op
controls
control_logic
ALUControl
25–21
20–16
15–11
R1
R2
W1
WD
RD1
RD2
register_file
ALUControl
function
In1
In2
out
ALU
D
Q
PC
C
Q
addr
Instr
Instr_mem
In1
4
In2
out
pc_adder

Combining datapaths

We do not want computer that can handle R-type *or* I-type instructions – should handle R-type *and* I-type instructions

When circuit sees R-type opcode, does one thing, and does a different thing when instruction has I-type opcode

As always with circuits, we cannot move around wires depending on inputs. When we want to behave differently based on inputs, we use…

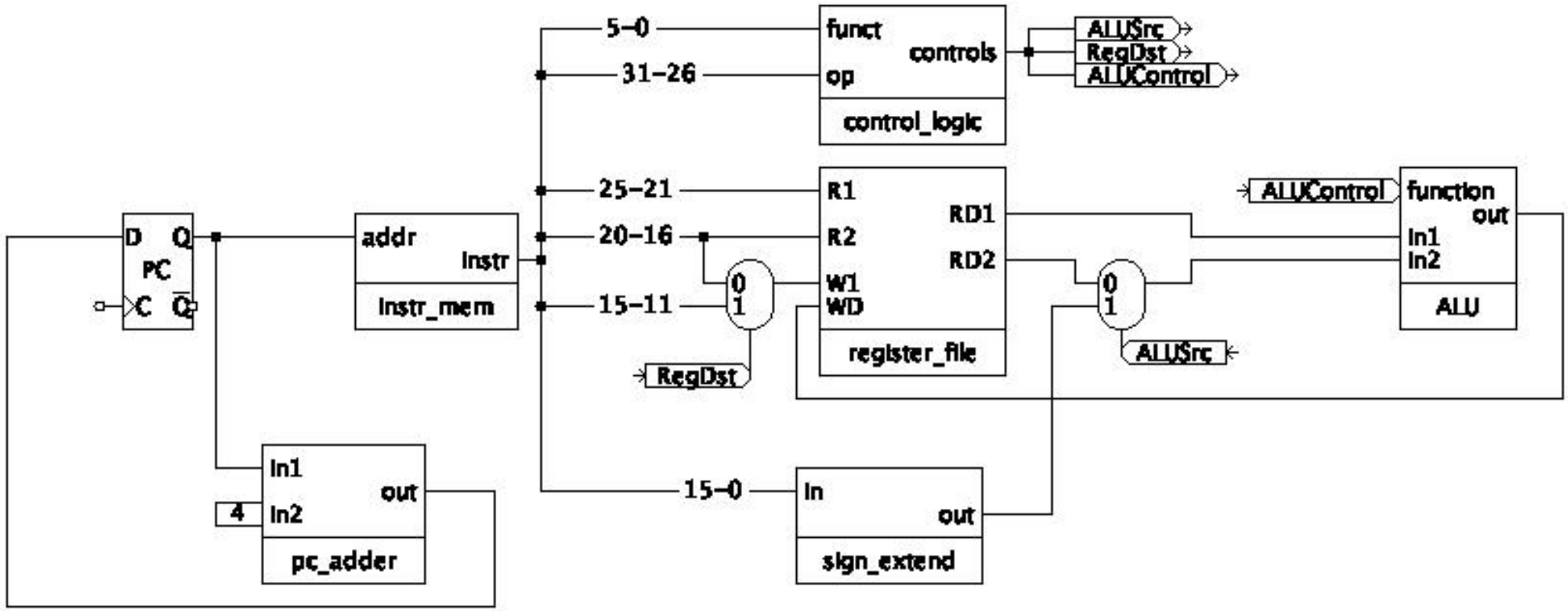
Combining datapaths

When we want to behave differently based on inputs, we use muxes!

Wherever discrepancy exists between instruction types, hook up circuit both ways and use mux to choose between them

Control logic no longer just for determining ALU operation – now responsible for controlling muxes as well

We continue to treat control logic as black box for now to simplify design, but will peek inside the box later


5–0
31–26
funct
op
controls
control_logic
ALUSrc
RegDst
ALUControl
25–21
20–16
15–11
R1
R2
W1
WD
RD1
RD2
register_file
0
1
RegDst
ALUControl
function
out
In1
In2
ALU
0
1
ALUSrc
D
Q
PC
C
Q
addr
Instr
Instr_mem
In1
4
In2
out
pc_adder
15–0
In
out
sign_extend

Muxes

`RegDst`:

- R-type instructions: rd is destination (bits 11 – 15)
- I-type instructions: rt is destination (bits 16 – 20)

`ALUSrc`:

- R-type instructions: second operand comes from register
- I-type instructions: second operand comes from immediate

R-type and I-type

By adding two muxes and some control logic, were able to combine previous circuits in straightforward way

New circuit correctly handles both types of instructions

All choices made automatically by examining instruction