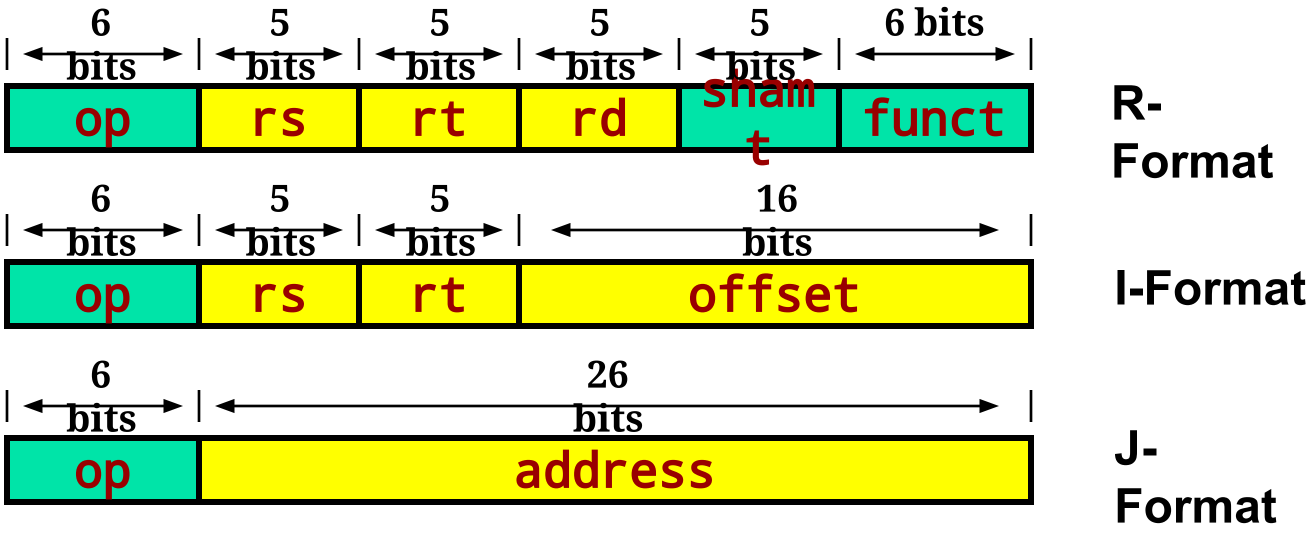
EE204: Computer Architecture

Chapter 4 (4.1 – 4.3)

Implementing MIPS

- We're ready to look at an implementation of the MIPS instruction set
- Simplified to contain only
  - arithmetic-logic instructions: `add, sub, and, or, slt`
  - memory-reference instructions: `lw, sw`
  - control-flow instructions: `beq, j`



6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits

op | rs | rt | rd | shamt | funct

R-Format

6 bits | 5 bits | 5 bits | 16 bits

op | rs | rt | offset

I-Format

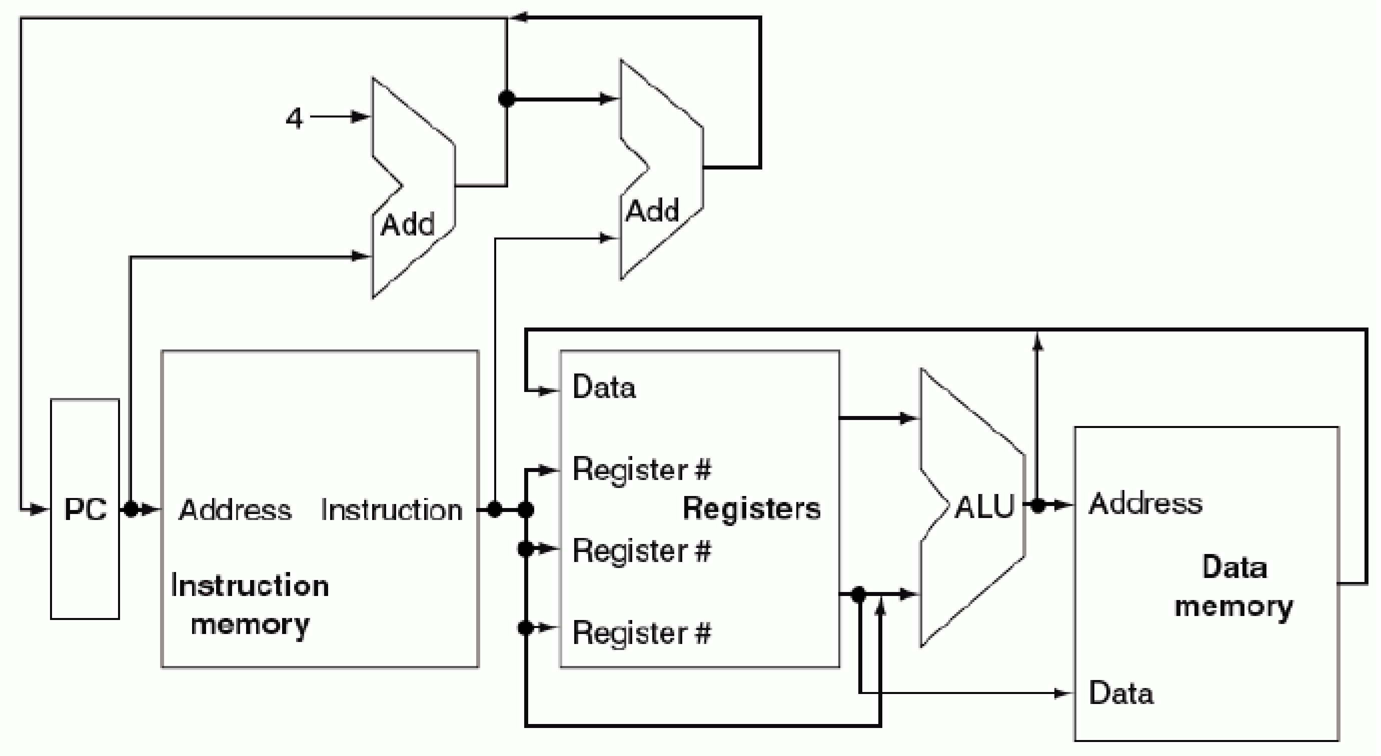
6 bits | 26 bits

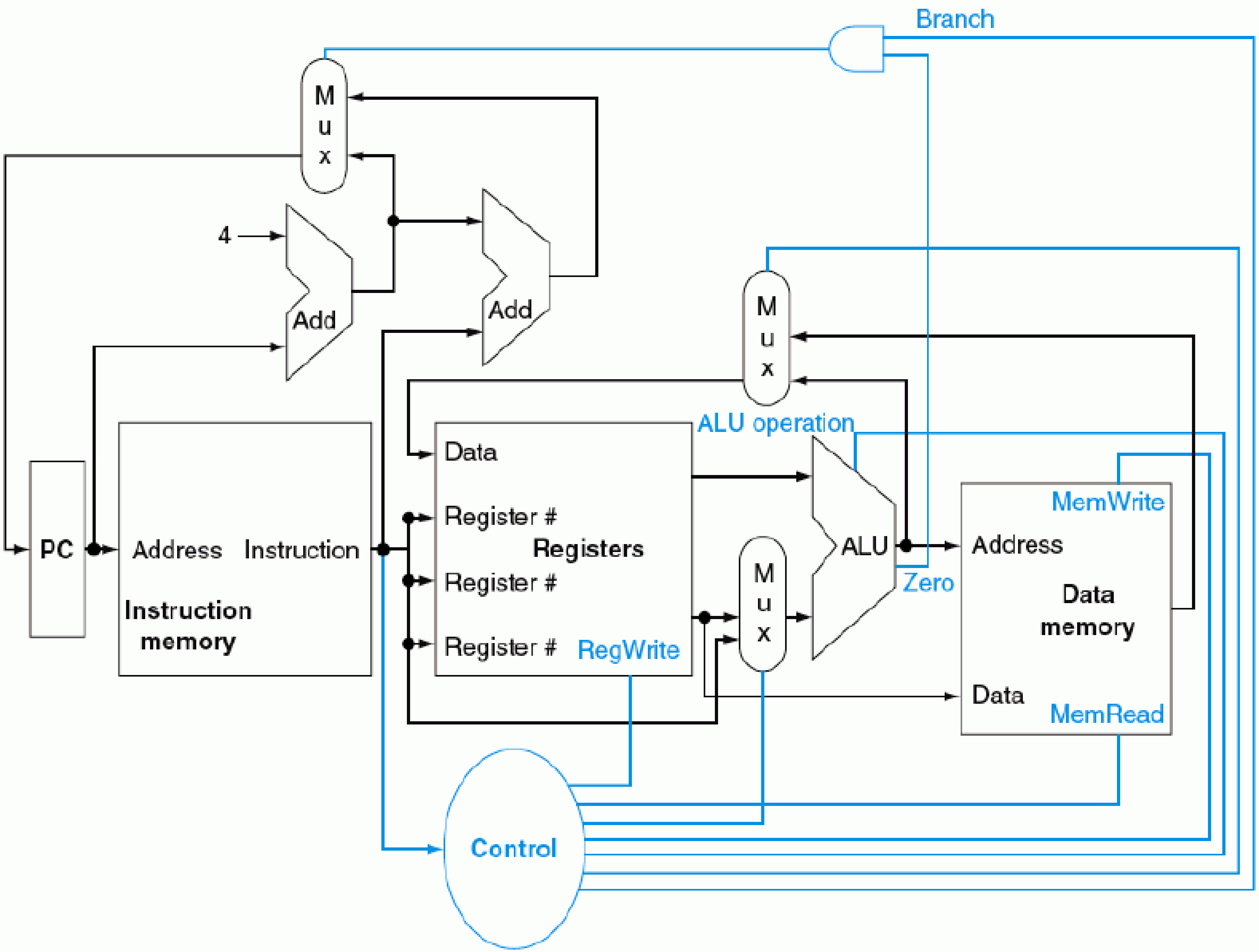
op | address

J-Format

Implementing MIPS: the Fetch/Execute Cycle

- High-level abstract view of *fetch/execute* implementation
  - use the program counter (PC) to read instruction address
  - *fetch* the instruction from memory and increment PC
  - use fields of the instruction to select registers to read
  - *execute* depending on the instruction
  - repeat…




Branch
M
u
x
4
Add
Add
M
u
x
ALU operation
Data
MemWrite
PC
Address
Instruction
Register #
Registers
Register #
Register #
RegWrite
Instruction memory
M
u
x
ALU
Zero
Address
Data memory
Data
MemRead
Control

Overview: Processor Implementation Styles

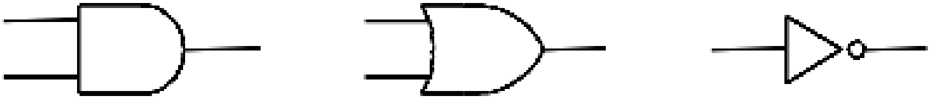
- Single Cycle
  - perform each instruction in 1 clock cycle
  - clock cycle must be long enough for slowest instruction; therefore,
  - disadvantage: only as fast as slowest instruction
- Multi-Cycle
  - break fetch/execute cycle into multiple steps
  - perform 1 step in each clock cycle
  - advantage: each instruction uses only as many cycles as it needs
- Pipelined
  - execute each instruction in multiple steps
  - perform 1 step / instruction in each clock cycle
  - process multiple instructions in parallel – assembly line

Logic Elements

- Two types of functional elements in the hardware:
  - elements that *operate on* data (called *combinational elements*)
  - elements that *contain* data (called *state* or *sequential elements*)

Combinational Elements

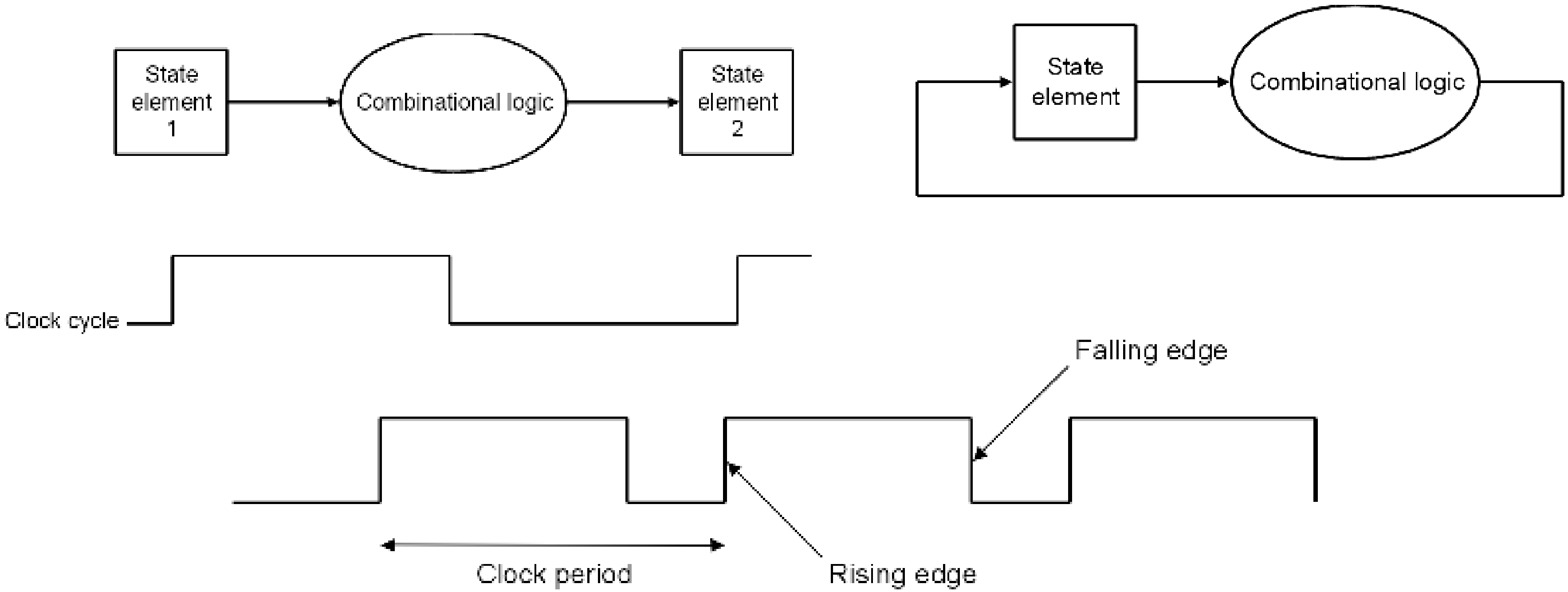
- Works as an *input* ⇒ *output function*, e.g., ALU
- Combinational logic *reads input data from one register and writes output data to another*, or same, register
  - *read/write happens in a single cycle* – combinational element *cannot store data* from one cycle to a future one



Combinational logic hardware units

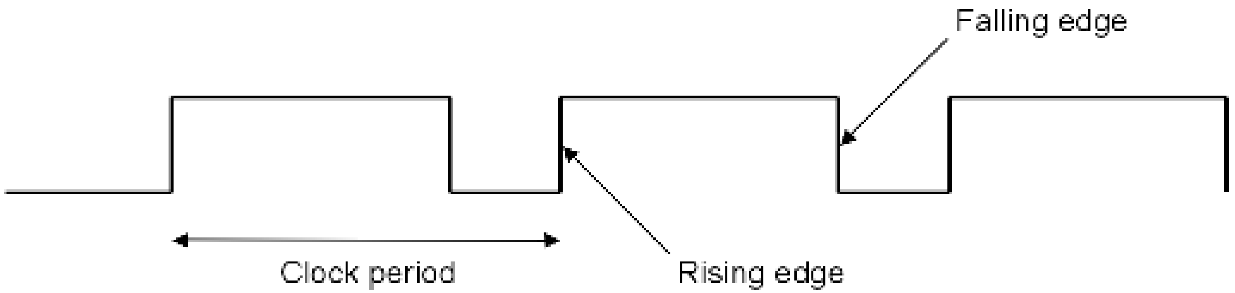
State Elements

- State elements contain *data* in internal storage, e.g., *registers* and*memory*



State Elements

- Clocks are used in *synchronous* logic to determine *when* a state element is to be updated
  - in *level-triggered* clocking methodology either the state changes only when the clock is high or only when it is low (technology-dependent)

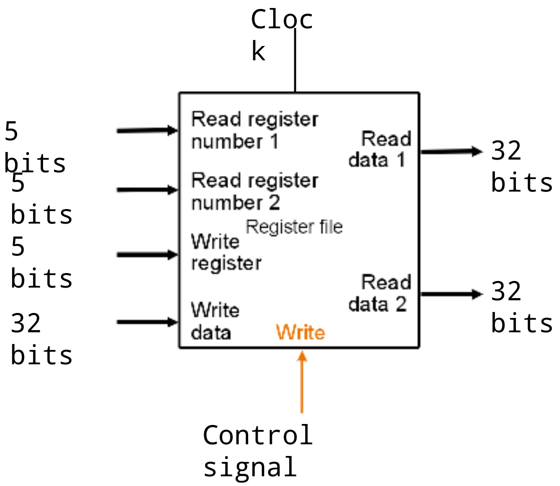


  - in *edge-triggered* clocking methodology either the *rising edge* or *falling edge* is active (depending on technology) – i.e., states change only on rising edges or only on falling edge

- Latches are level-triggered
- Flipflops are edge-triggered

State Elements (Register File)

- Registers are implemented with arrays of D-flipflops



Clock

5 bits

5 bits

5 bits

32 bits

32 bits

32 bits

Control signal

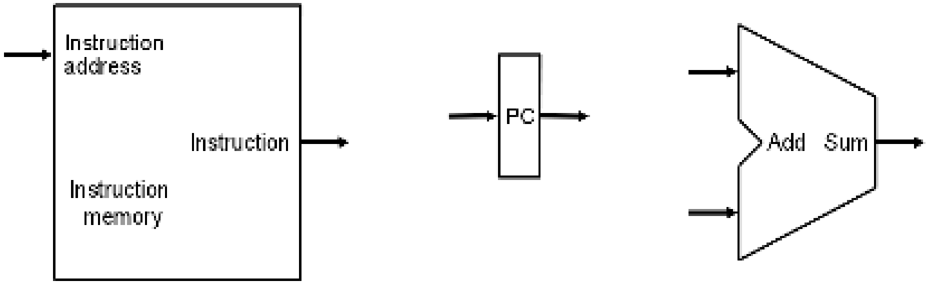
Register file with two read ports and one write port

Single-cycle Implementation of MIPS

- Our first implementation of MIPS will use a *single* long clock cycle for every instruction
- Every instruction begins on one up (or, down) clock edge and ends on the next up (or, down) clock edge
- This approach is *not practical* as it is much slower than a *multicycle* implementation where different instruction classes can take different numbers of cycles
  - in a single-cycle implementation every instruction must take the same amount of time as the slowest instruction
  - in a multicycle implementation this problem is avoided by allowing quicker instructions to use fewer cycles
- Even though the single-cycle approach is not practical it is simple and useful to understand first

- *Note* : we shall implement `jump` at the very end

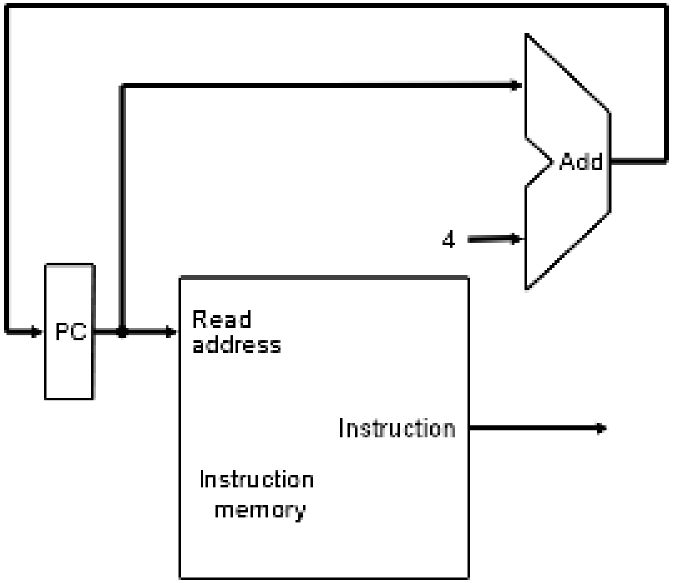
Datapath: Instruction Store/ Fetch & PC Increment



a. Instruction memory

b. Program counter

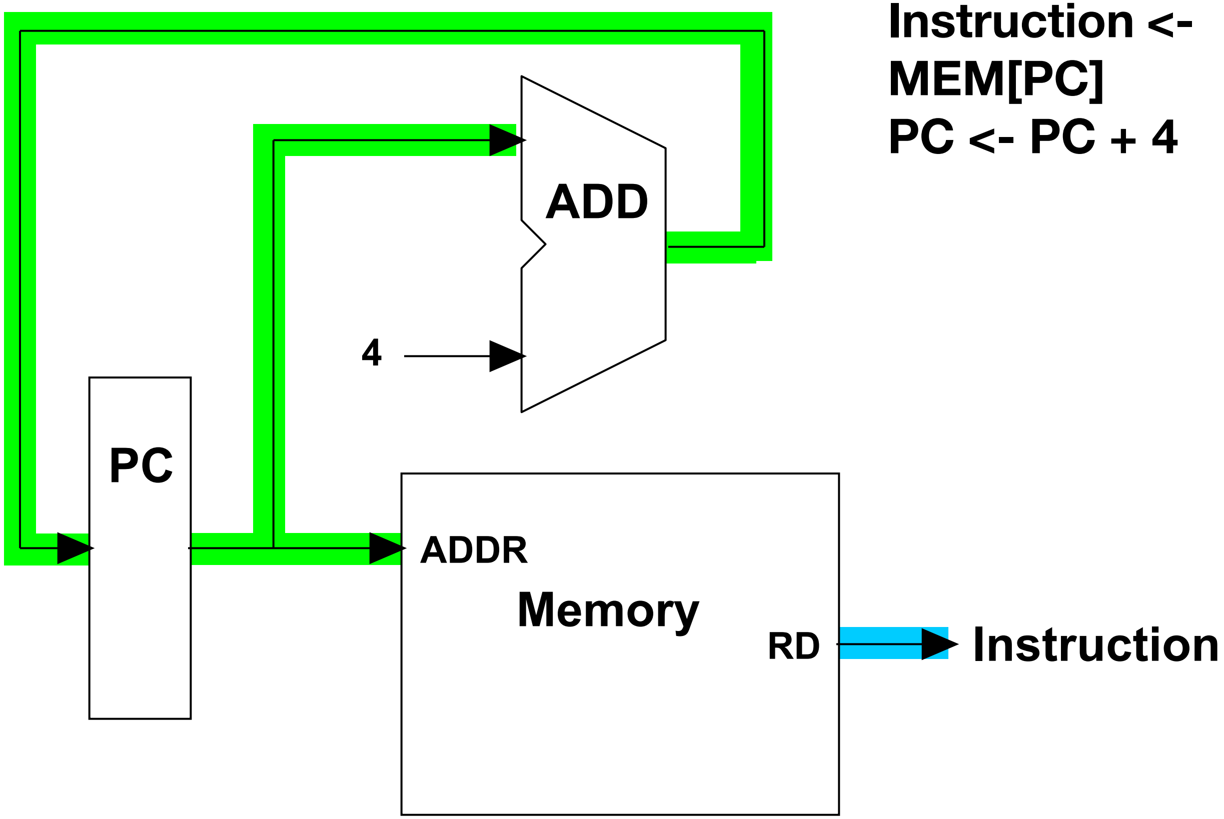
c. Adder



Three elements used to store and fetch instructions and increment the PC

**Datapat
h**

Animating the Datapath

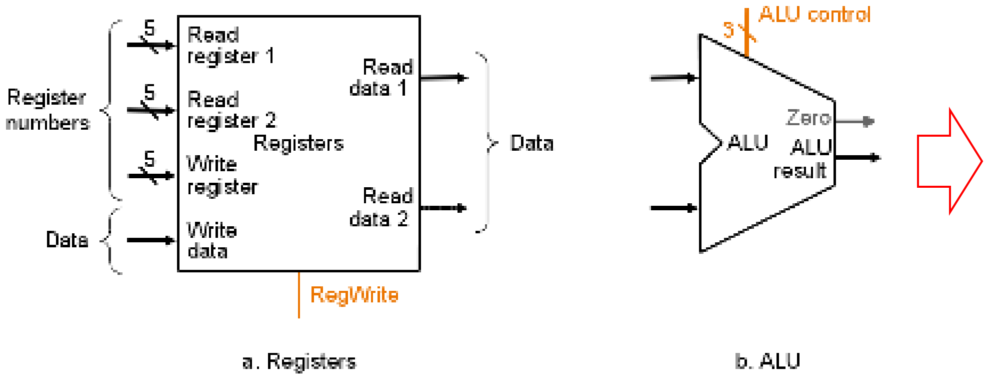


Instruction <-
MEM[PC]
PC <- PC + 4

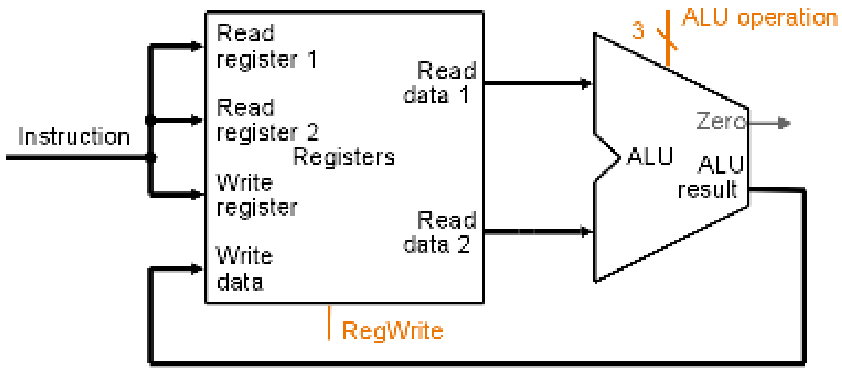
Datapath: R-Type Instruction

6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
op	rs	rt	rd	shamt	funct

R-Format



Two elements used to implement R-type instructions



Datapat h

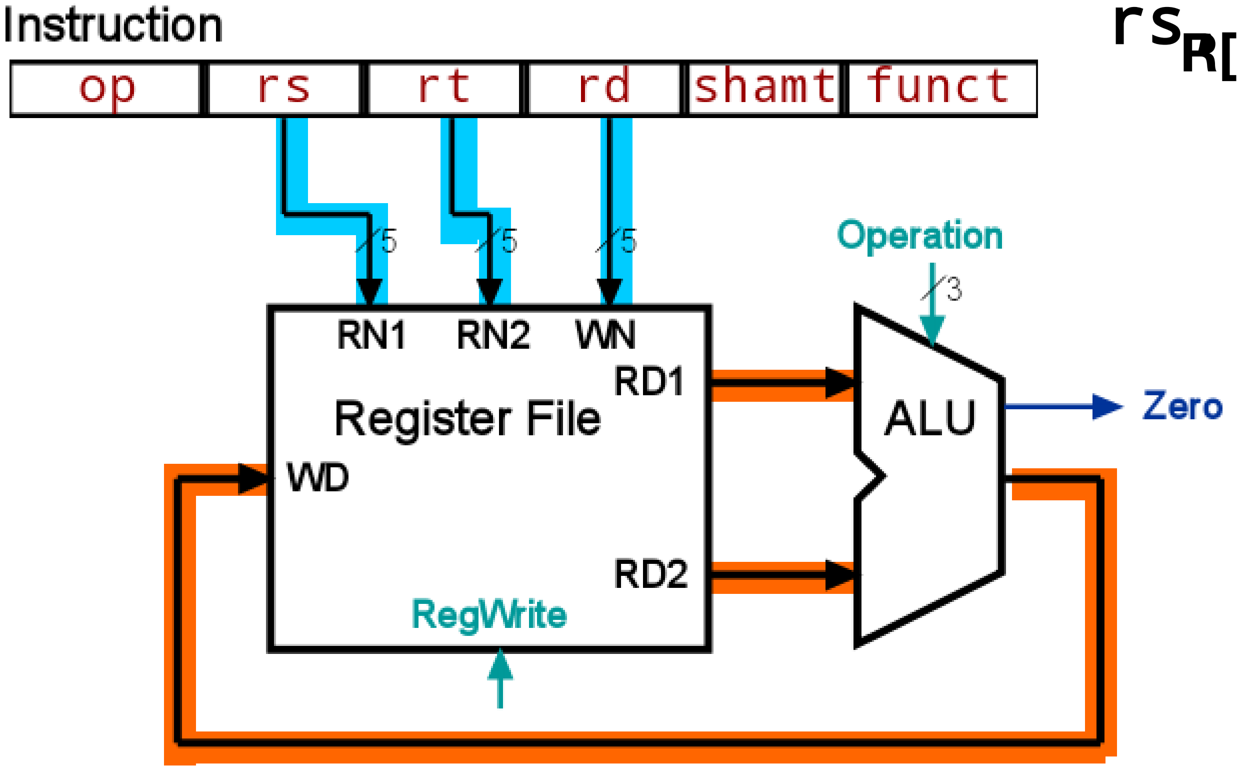
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
op	rs	rt	rd	shamt	funct

R-Format

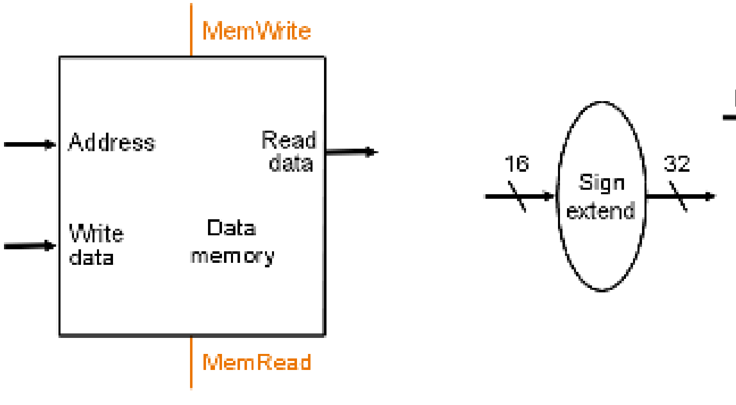
Animating the Datapath

```
add rd, rs, rt
```

R[rd] <- R[rs] + R[rt];

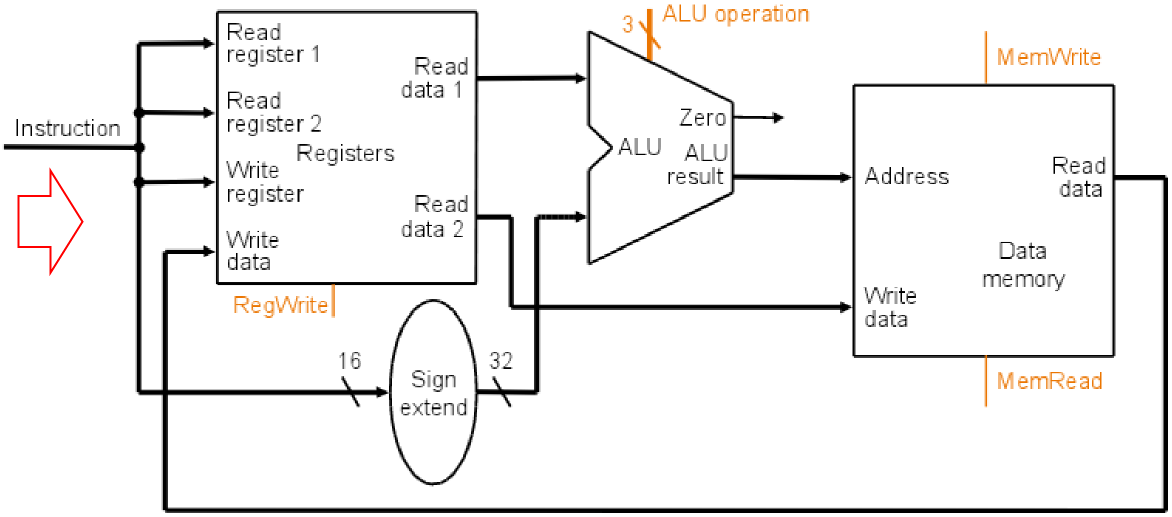


Datapath: Load/Store Instruction



a. Data memory unit

b. Sign-extension unit



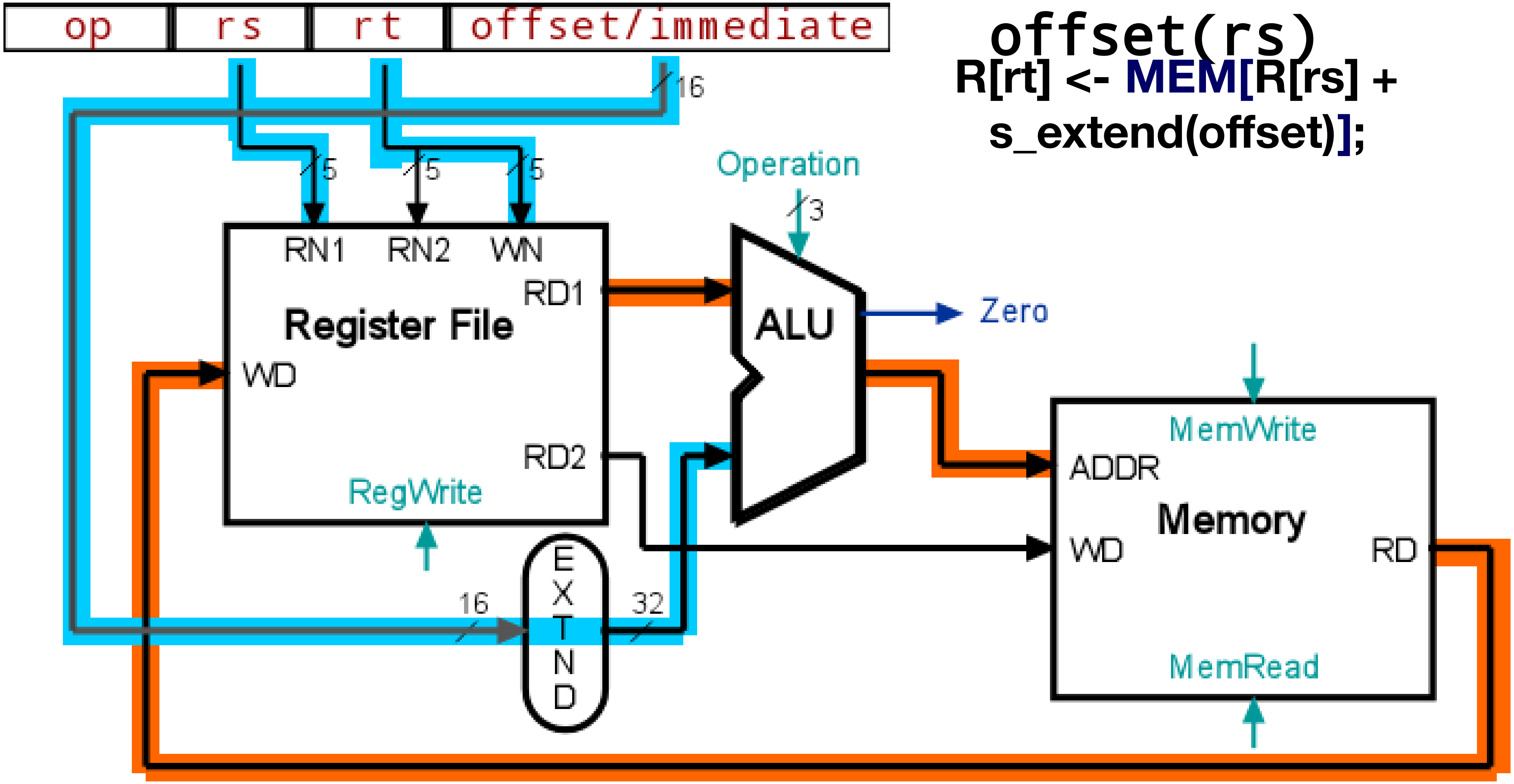
Two additional elements used
To implement load/stores

Datapat
h

Animating the Datapath

```
lw rt,
offset(rs)
```

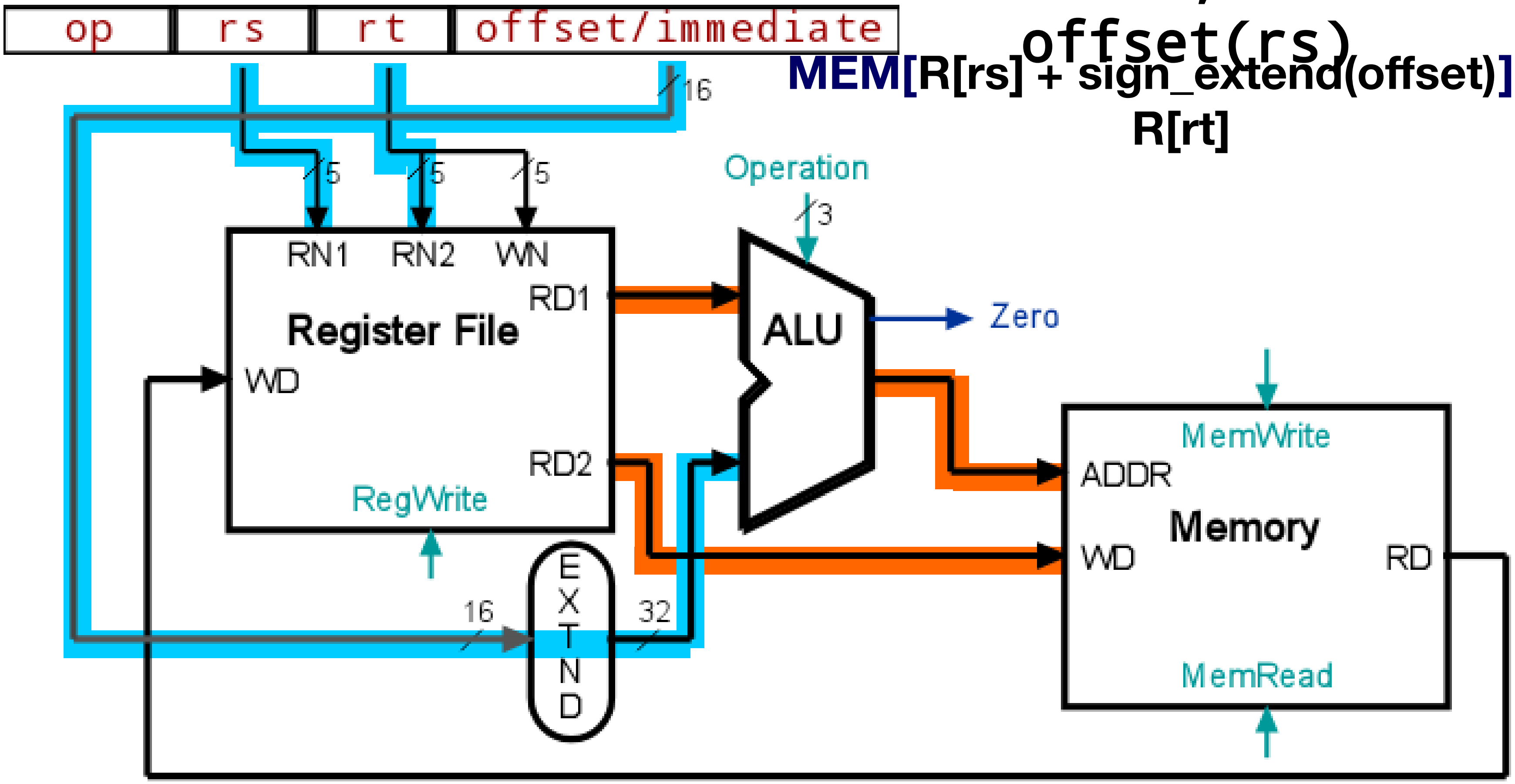
R[rt] <- MEM[R[rs] + s_extend(offset)];



Animating the Datapath

sw rt, offset(rs)

MEM[R[rs] + sign_extend(offset)] <- R[rt]

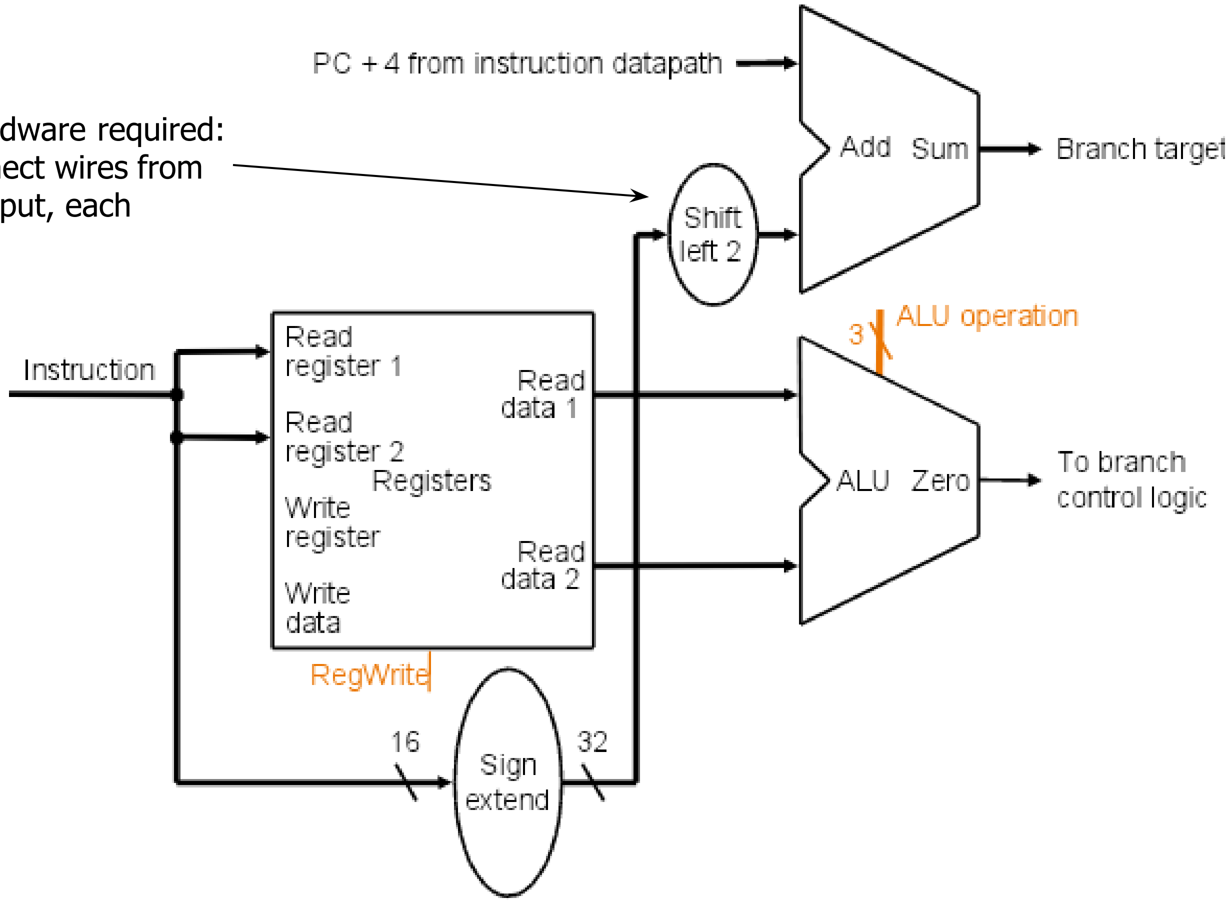


Datapath: Branch Instruction

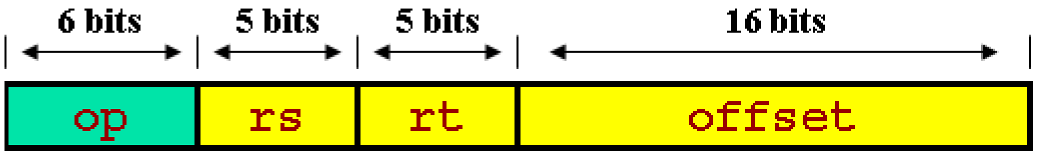
6 bits	5 bits	5 bits	16 bits
op	rs	rt	offset

I-Format

No shift hardware required: simply connect wires from input to output, each shifted
left 2 bits

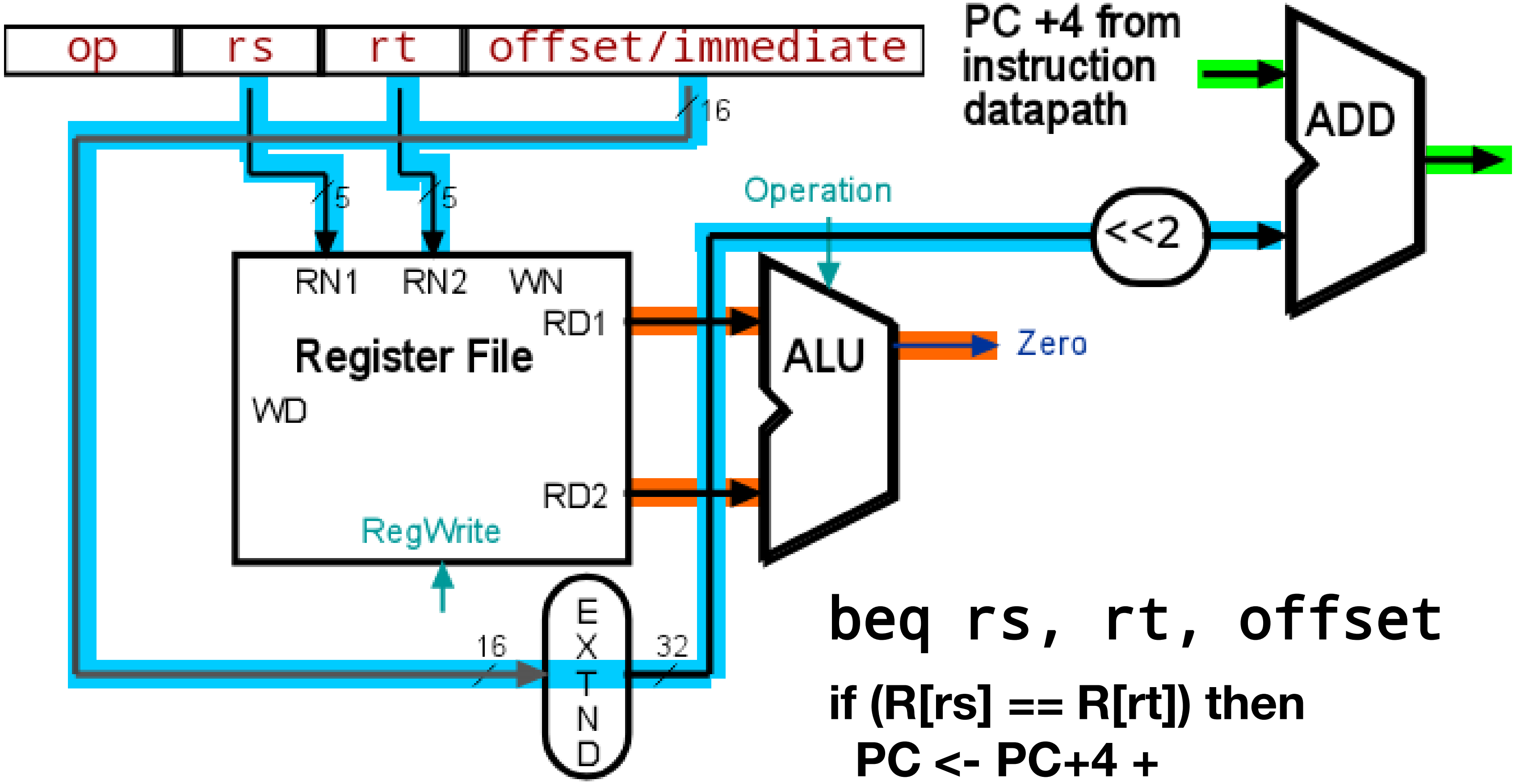


Datapat h



I-Format

Animating the Datapath

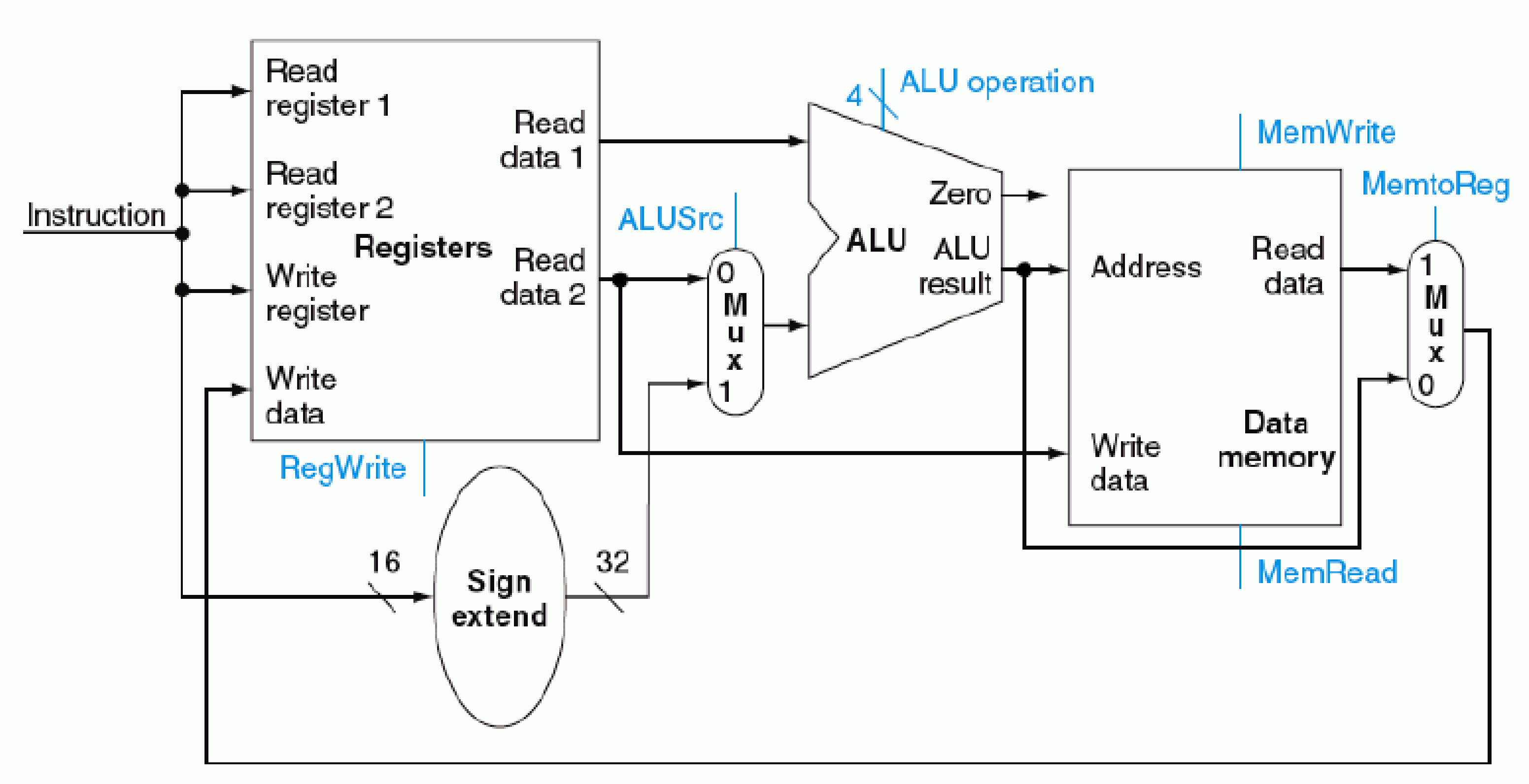


PC +4 from instruction datapath

beq rs, rt, offset

if (R[rs] == R[rt]) then
PC <- PC+4 +
s_extend(offset<<2)

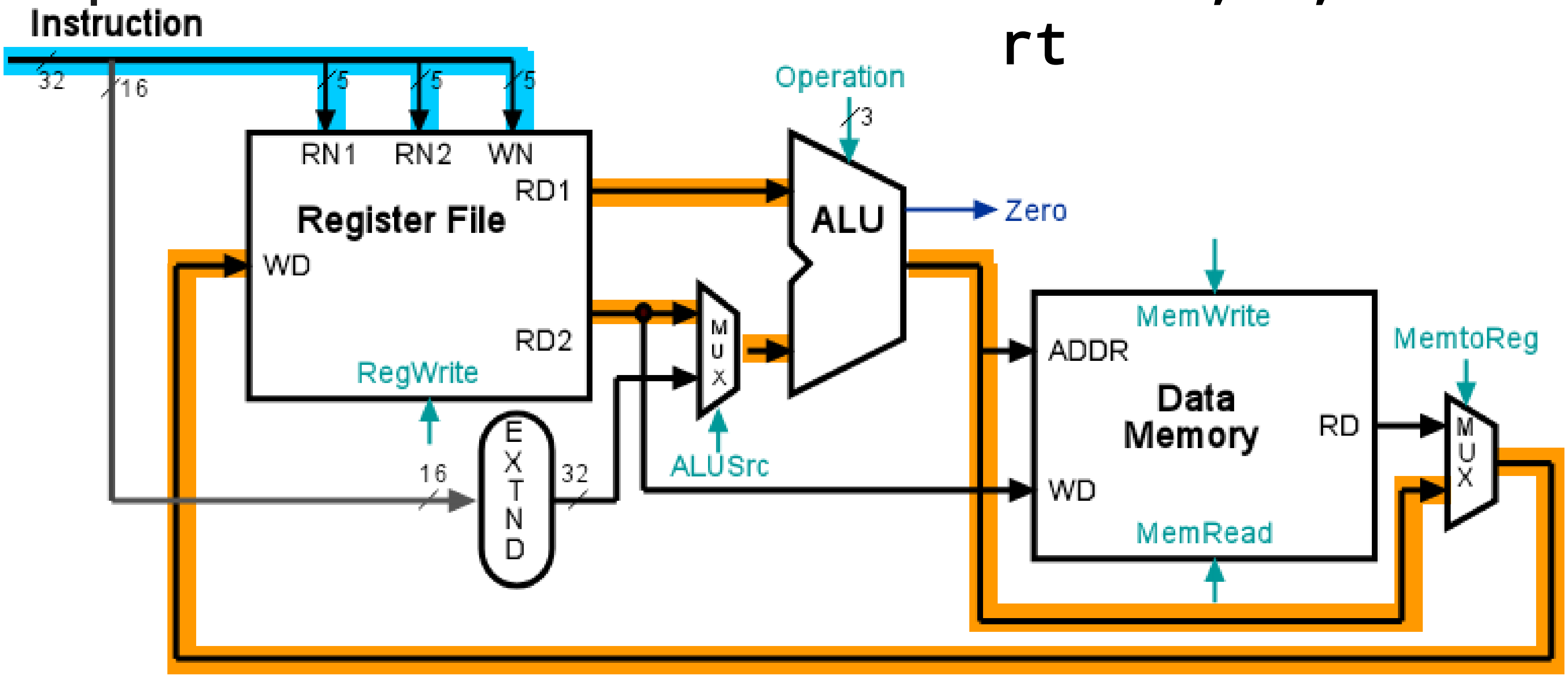
MIPS Datapath I: Single-Cycle



Combining the datapaths for R-type instructions and load/stores using two multiplexors

Animating the Datapath: R-type Instruction

```
add rd,rs,
rt
```

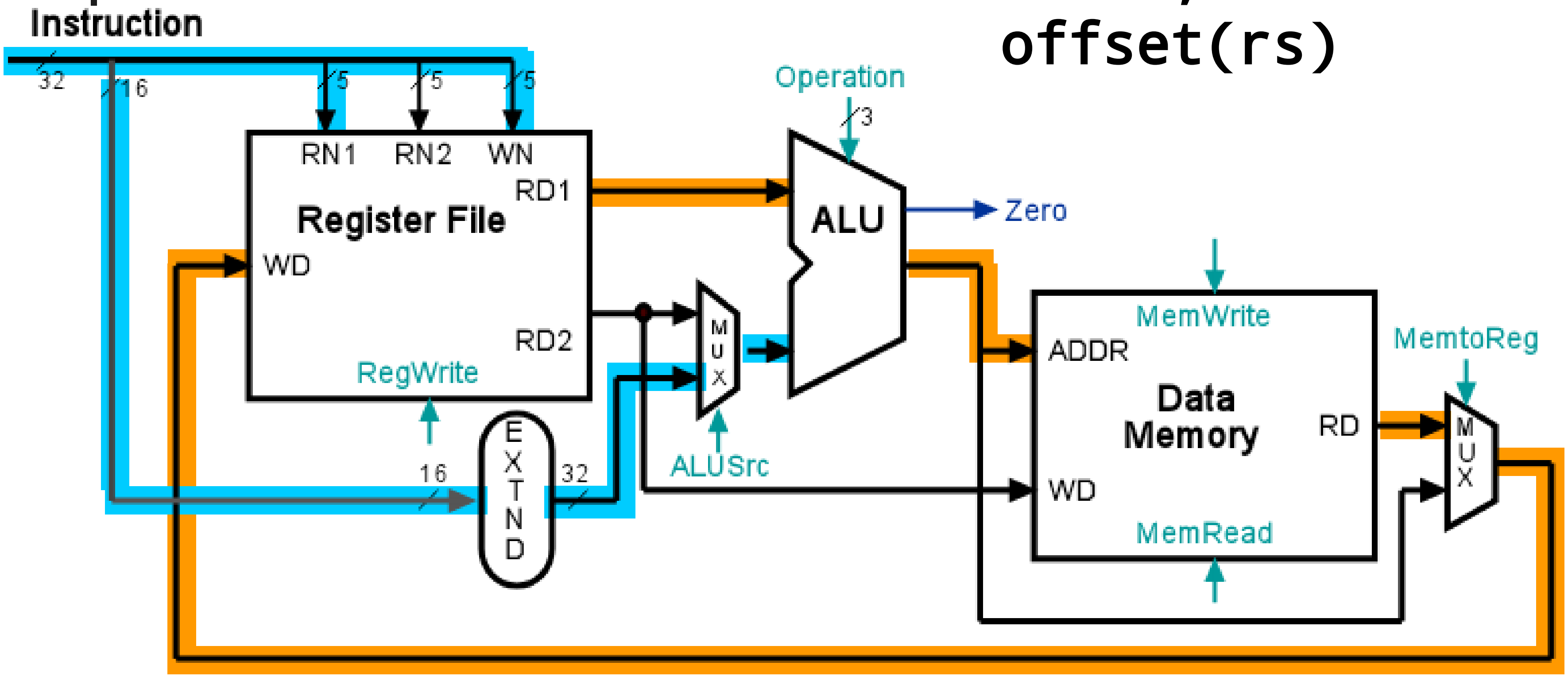


6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
op	rs	rt	rd	shamt	funct

R-Format

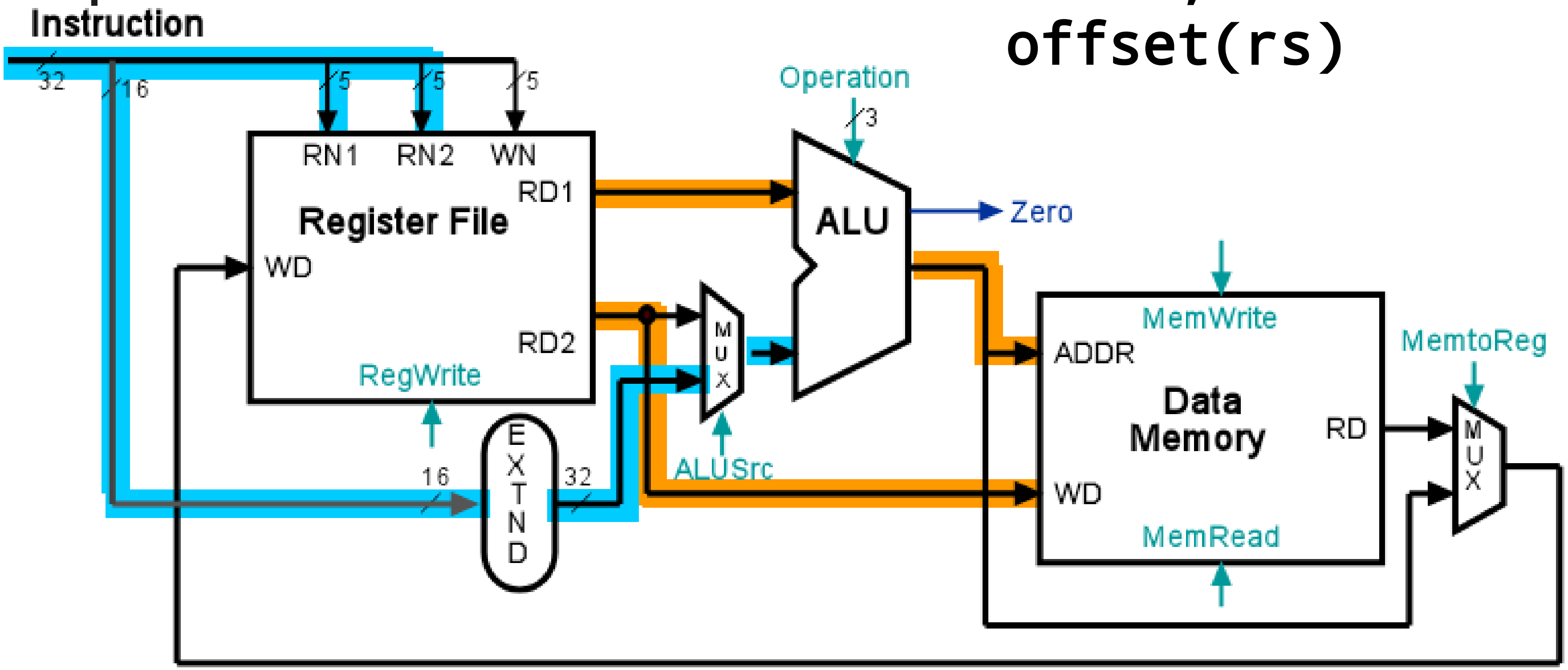
Animating the Datapath: Load Instruction

```
lw rt,
offset(rs)
```

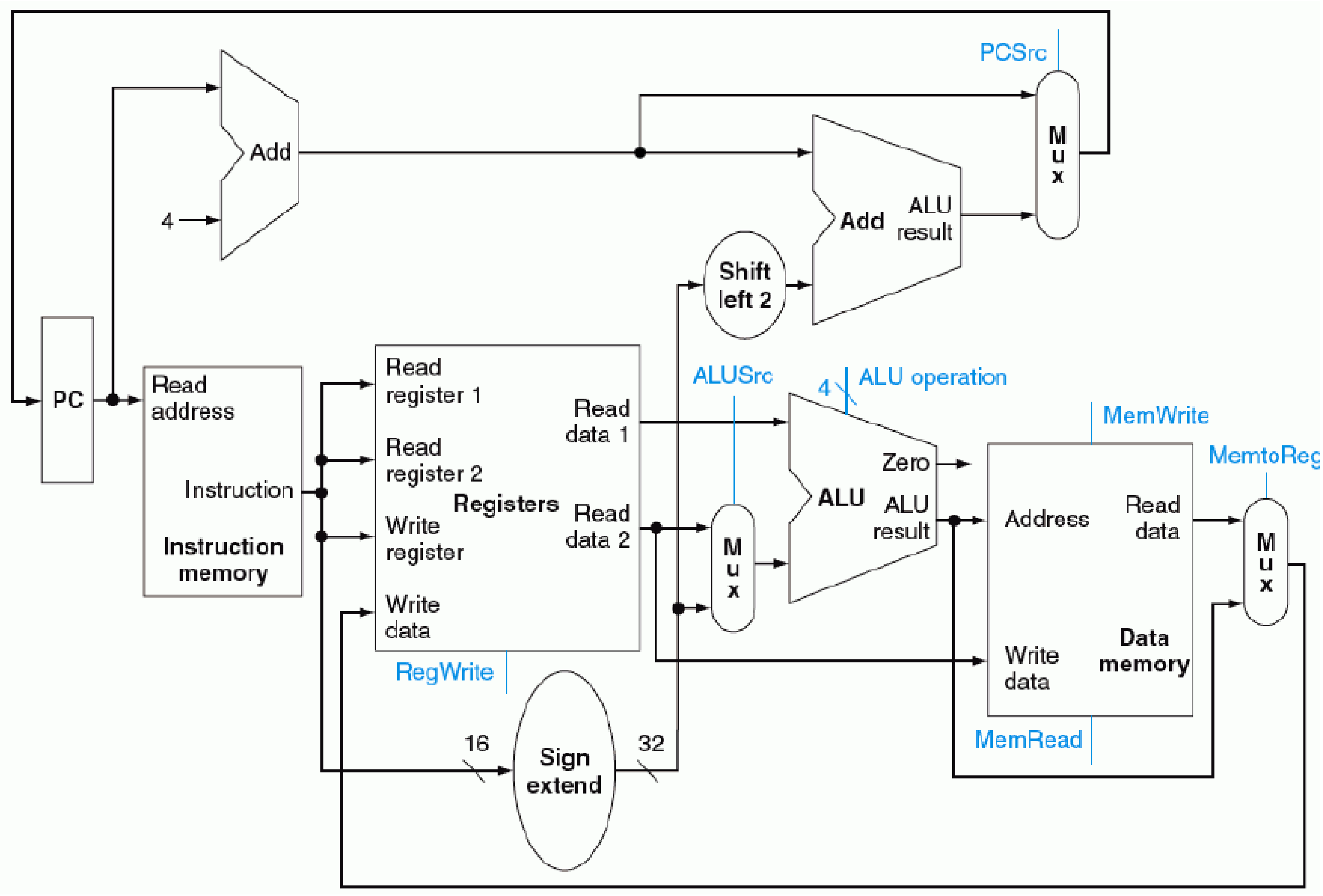


Animating the Datapath: Store Instruction

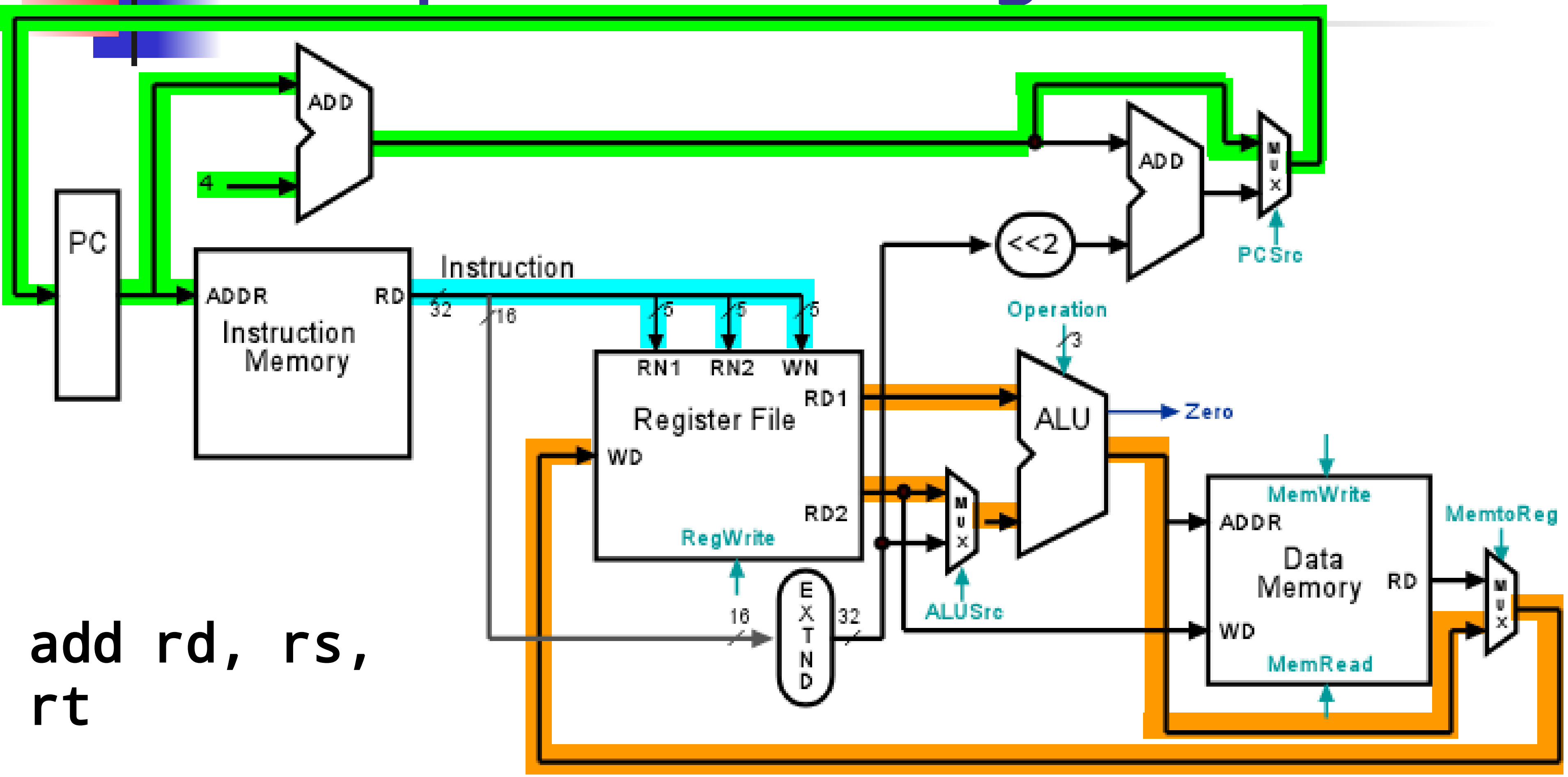
```
sw rt,
offset(rs)
```



MIPS Datapath II: Single-Cycle

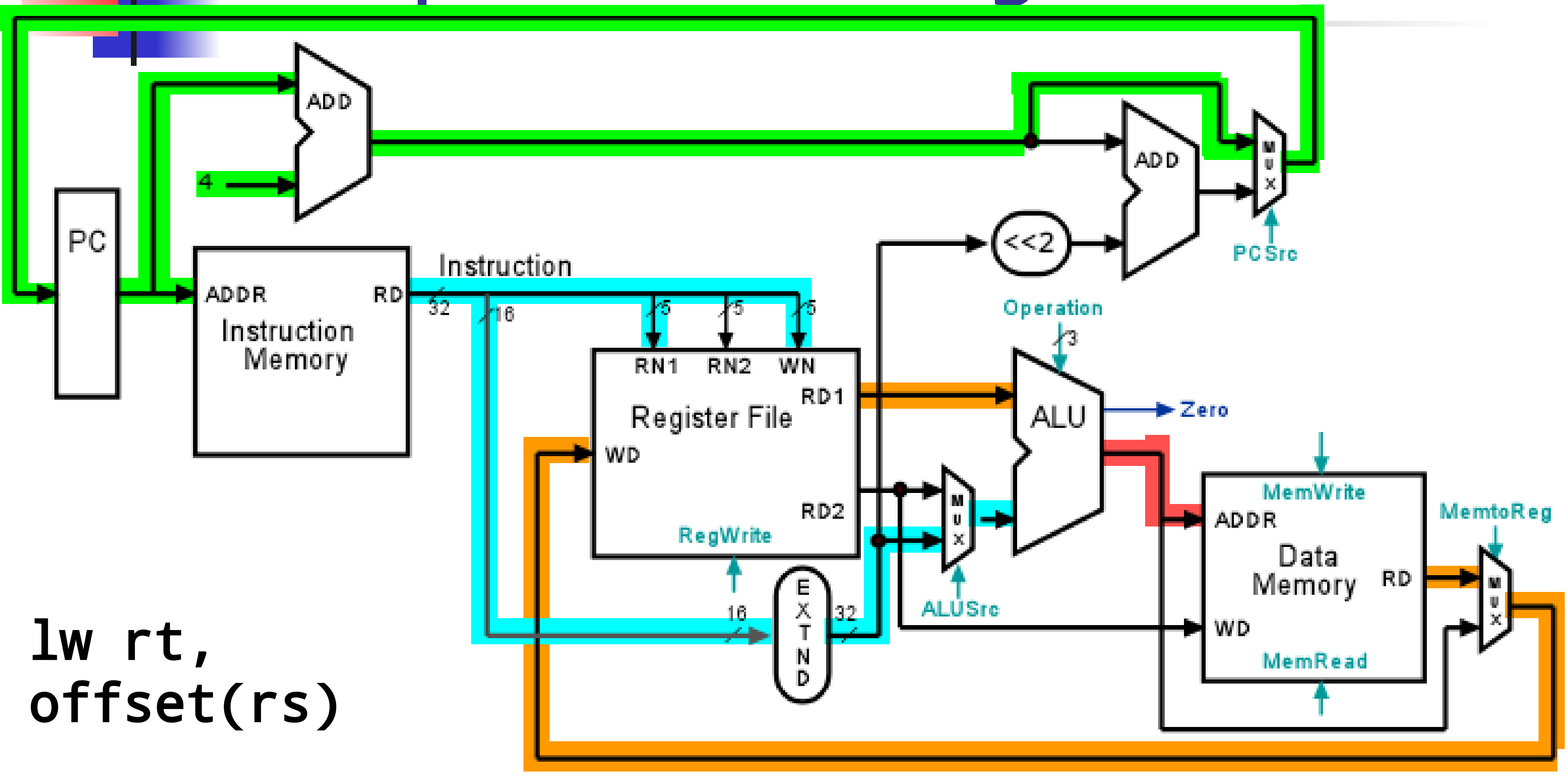


Datapath Executing add



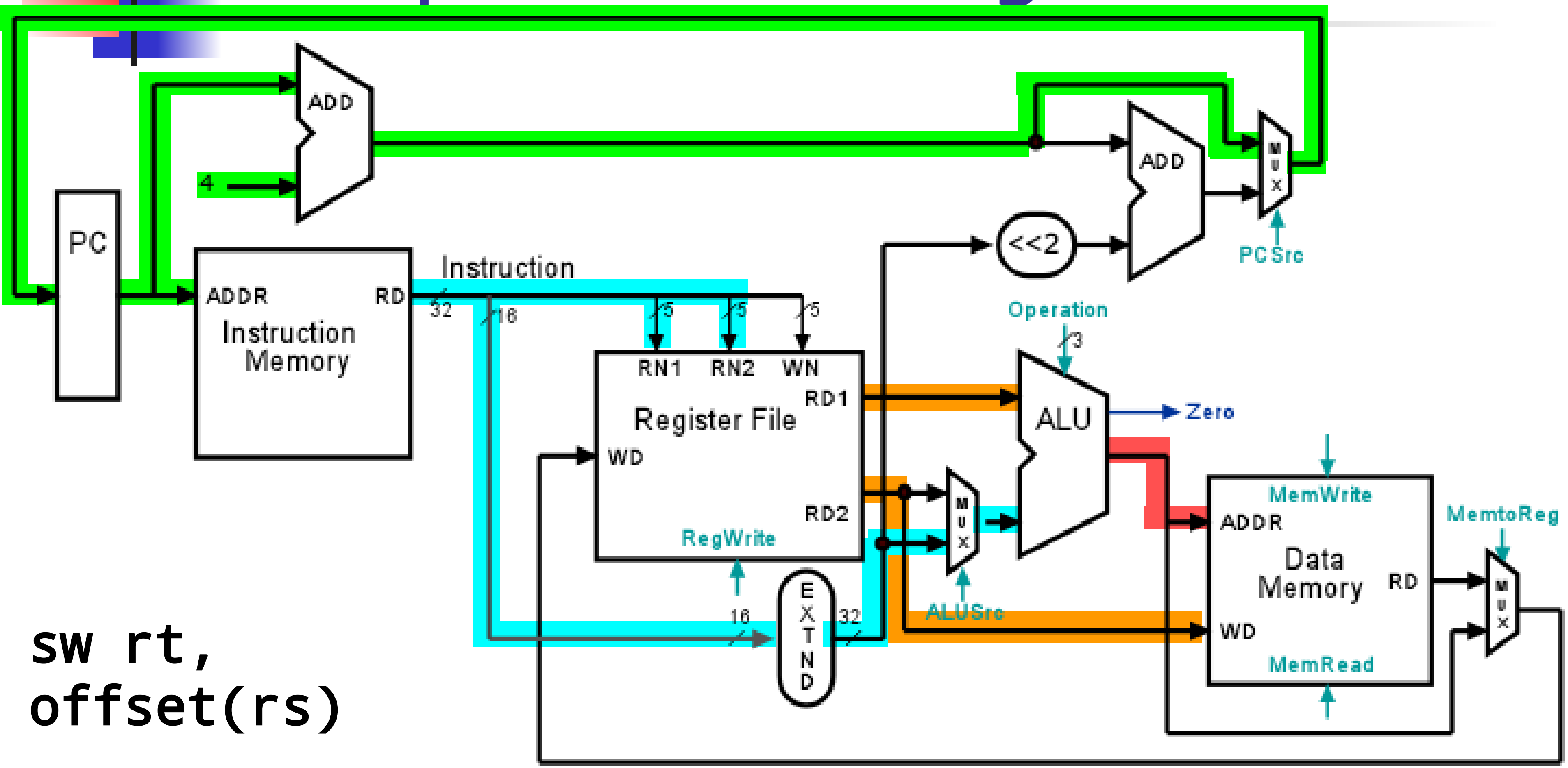
add rd, rs, rt

Datapath Executing `lw`



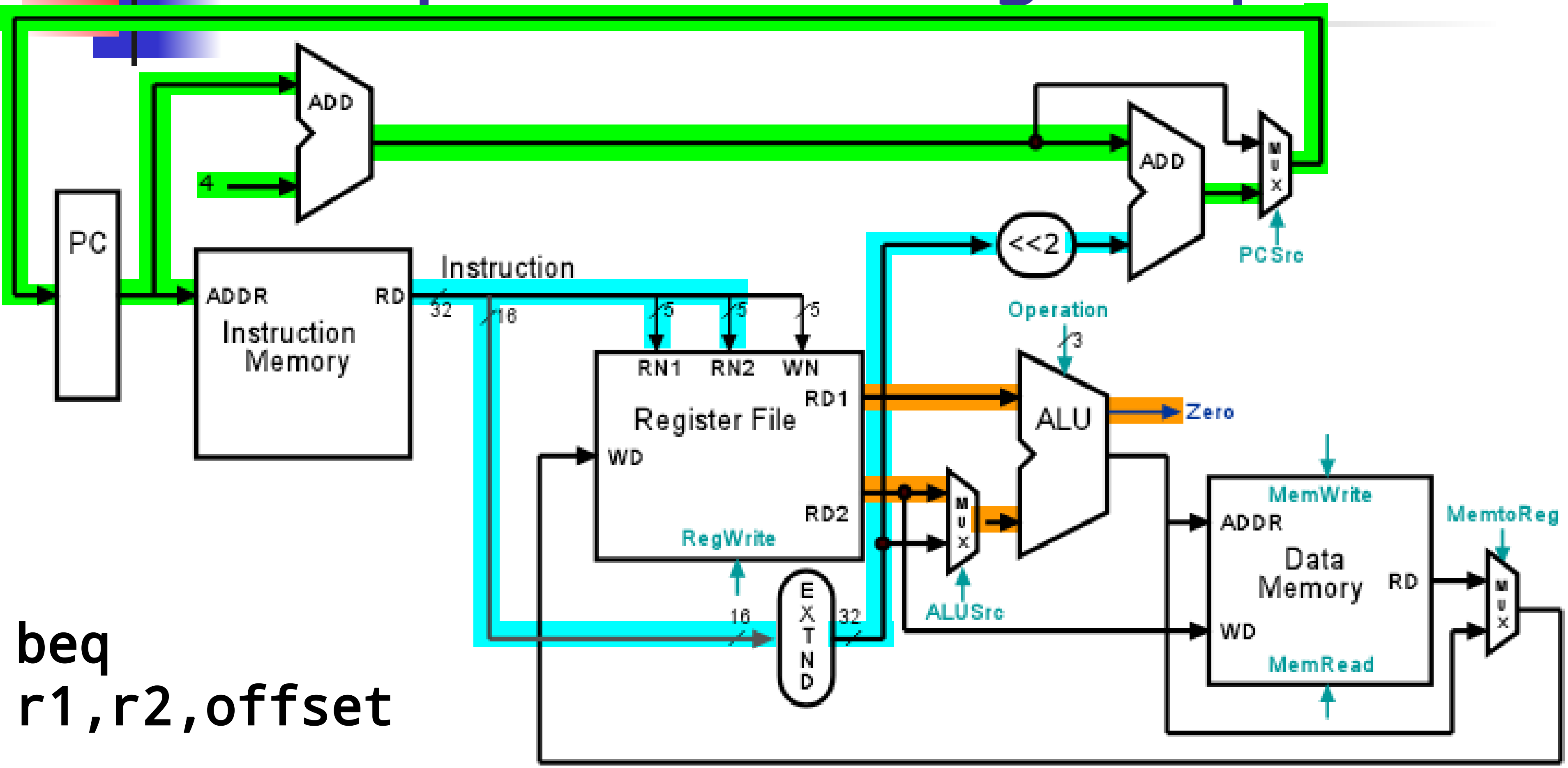
```
lw rt,
offset(rs)
```

Datapath Executing sw



sw rt, offset(rs)

Datapath Executing beq



beq r1,r2,offset