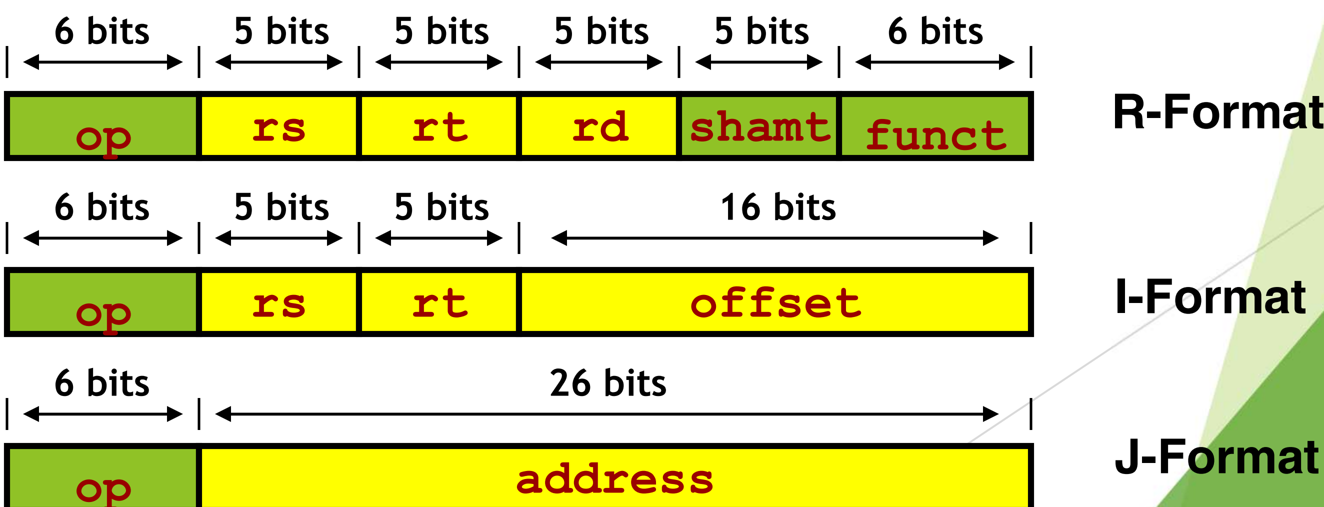
Ειδικά Θέματα Αρχιτεκτονικής και Προγραμματισμού Μικροεπεξεργαστών

Ενότητα 4: The Processor: Datapath and Control

Διδάσκων: Βαρτζιώτης Φώτιος
Τμήμα Πληροφορικής και Τηλεπικοινωνιών

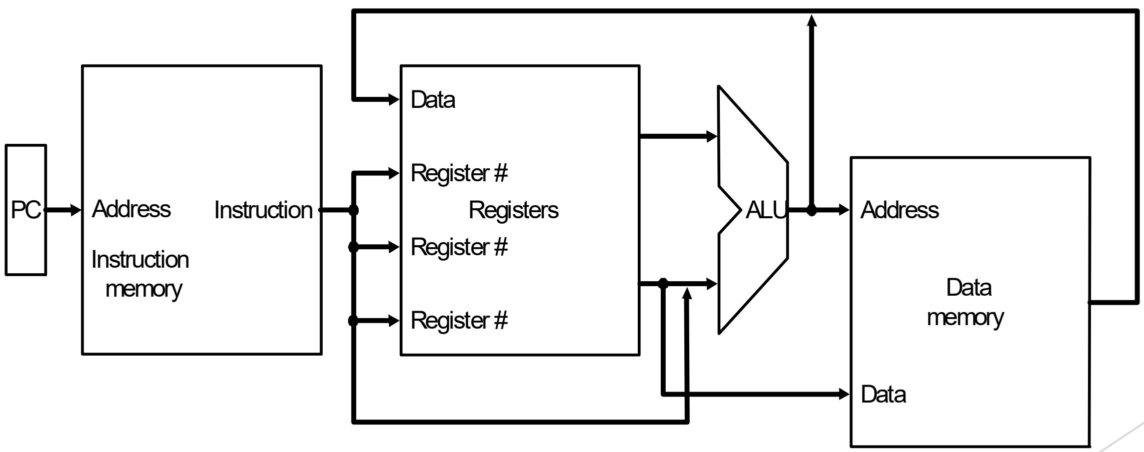
Implementing MIPS

- 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`



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…



Overview: Processor Implementation Styles

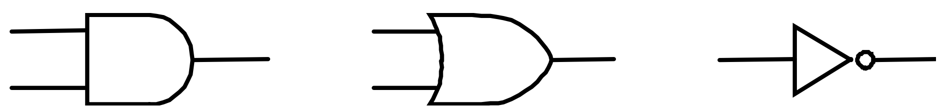
- Single Cycle
  - perform each instruction in 1 clock cycle
  - clock cycle must be long enough for slowest instruction; therefore,
  - disadvantage: only as fast as slowest instruction
- Multi-Cycle
  - break fetch/execute cycle into multiple steps
  - perform 1 step in each clock cycle
  - advantage: 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

Functional Elements

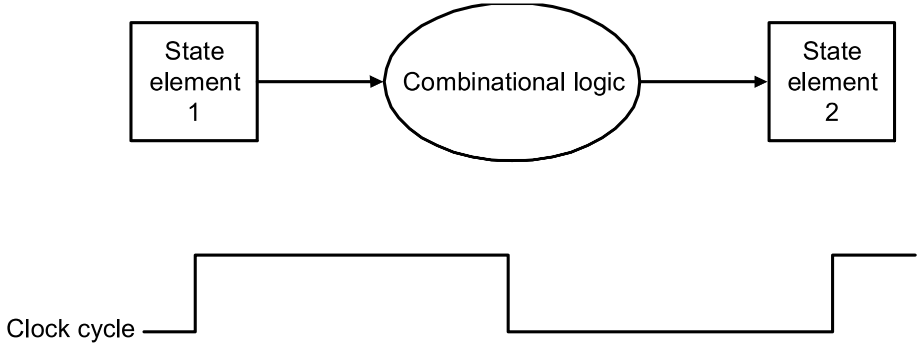
- Two types of functional elements in the hardware:
  - elements that *operate on* data (called *combinational elements*)
  - elements that *contain* data (called *state* or *sequential elements*)

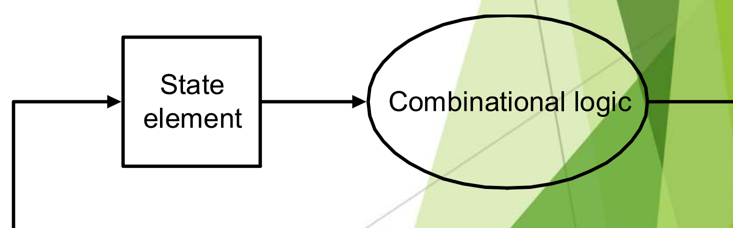
Combinational Elements

- Works as an *input* $\Rightarrow$ *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



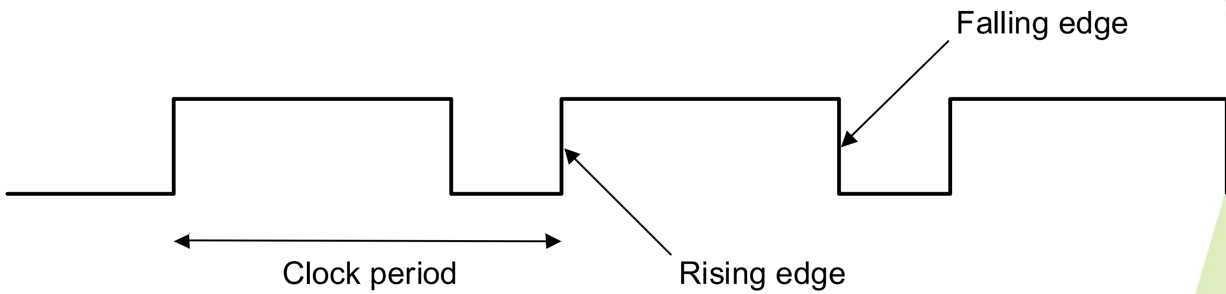


State Elements

- State elements contain *data* in internal storage, e.g., *registers* and *memory*
- All state elements together *define* the *state of the machine*
  - *What does this mean? Think of shutting down and starting up again…*
- *Flipflops* and *latches* are 1-bit state elements, equivalently, they are *1-bit memories*
- The *output*(s) of a flipflop or latch *always* depends on the bit value stored, i.e., its state, and can be called *1/0* or *high*/low or *true/false*
- The *input* to a flipflop or latch can change its state depending on whether it is clocked or not…

Synchronous Logic: Clocked Latches and Flipflops

- 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)

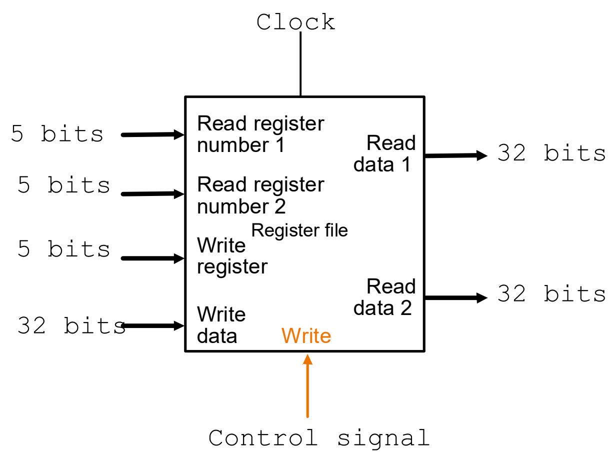


  - in *edge-triggered* clocking methodology either the *rising edge* or *falling edge* is active (depending on technology) – i.e., states change only on rising edges or only on falling edge

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

State Elements on the Datapath: Register File

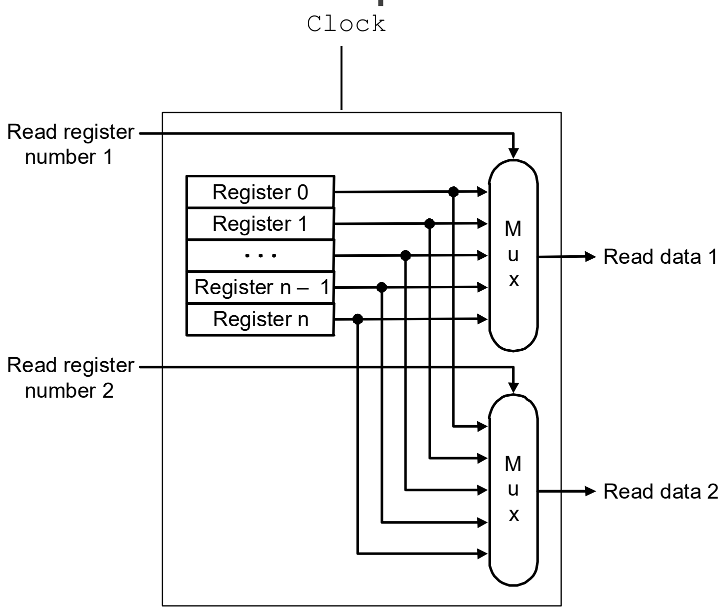
- Registers are implemented with arrays of D-flipflops

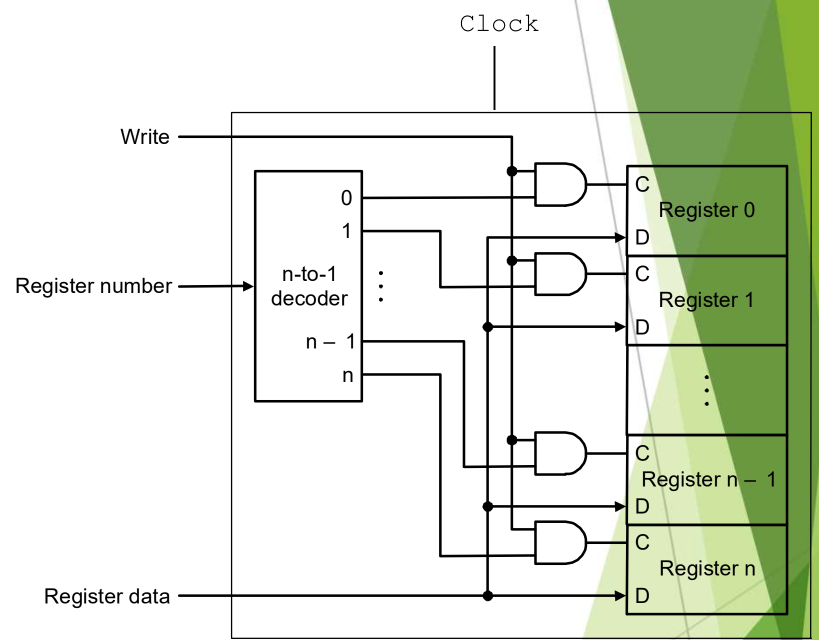


Register file with two read ports and one write port

State Elements on the Datapath: Register File

- Port implementation:





Read ports are implemented with a pair of multiplexors – 5 bit multiplexors for 32 registers

Write port is implemented using a decoder – 5-to-32 decoder for 32 registers. Clock is relevant to write as register state may change only at clock edge

VHDL

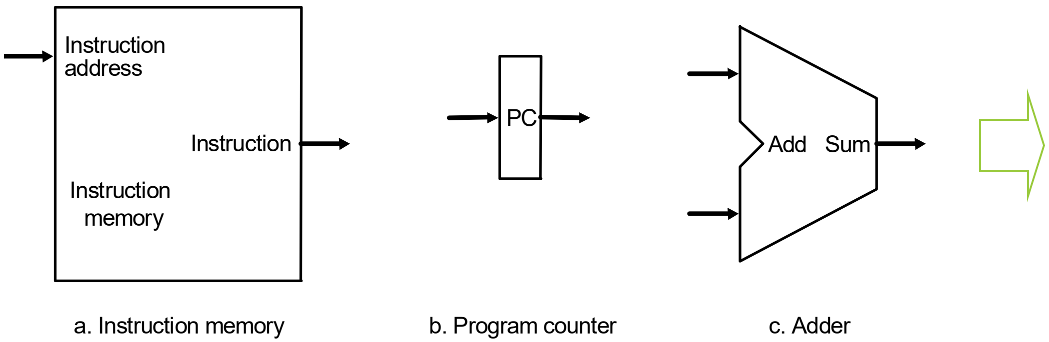
- All components that we have discussed – and shall discuss – can be fabricated using VHDL (or other HDL)
- Refer to VLSI design slides and examples

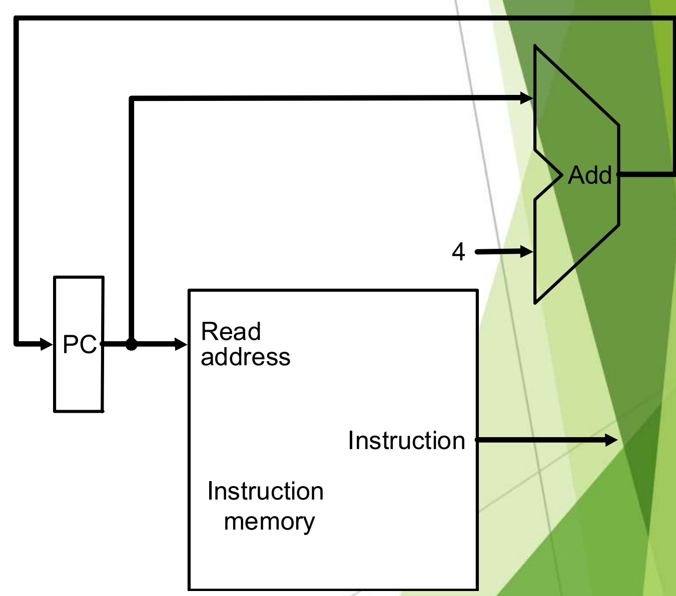
Single-cycle Implementation of MIPS

- 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

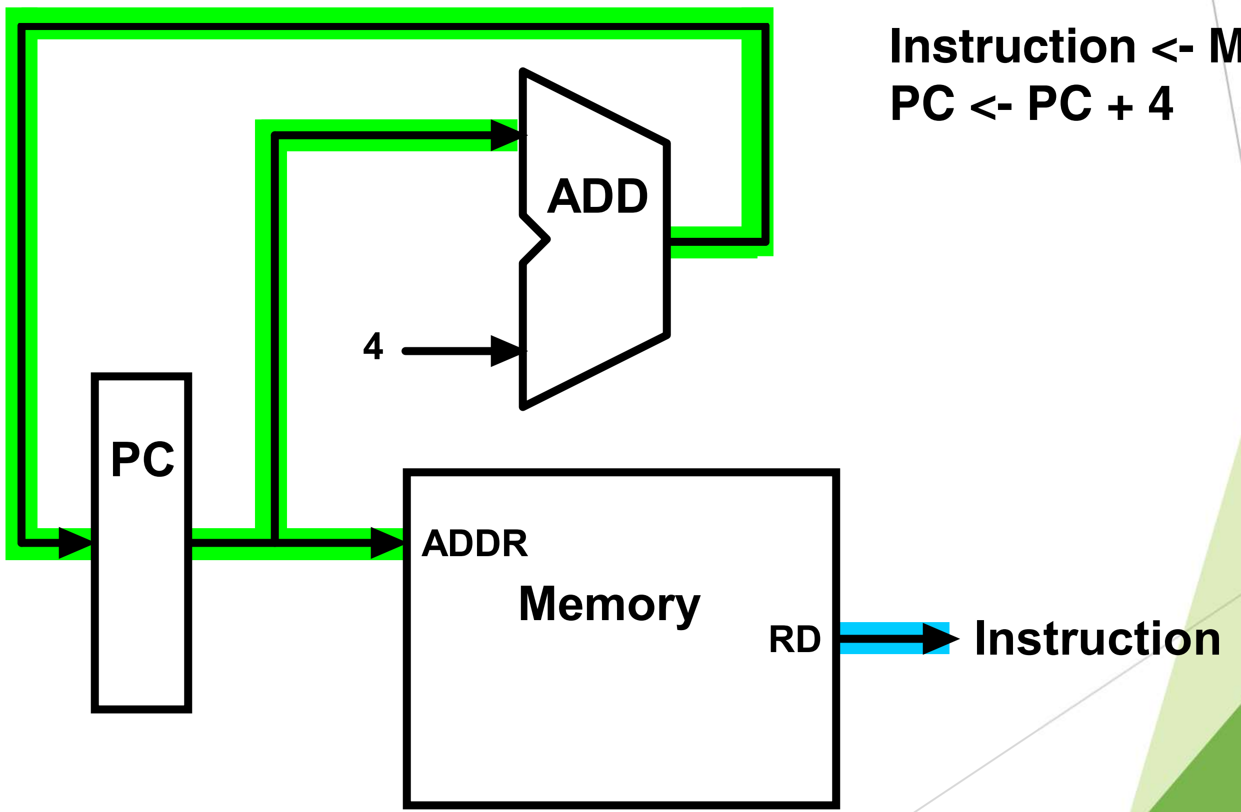




Three elements used to store and fetch instructions and increment the PC

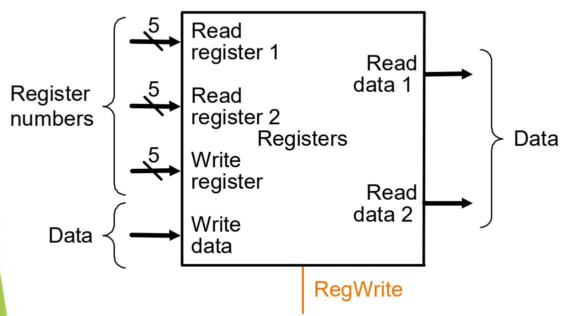
Datapath

Animating the Datapath

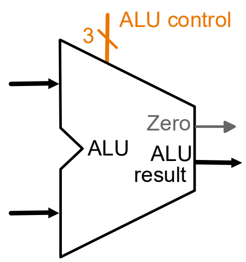


Instruction <- MEM[PC]
PC <- PC + 4

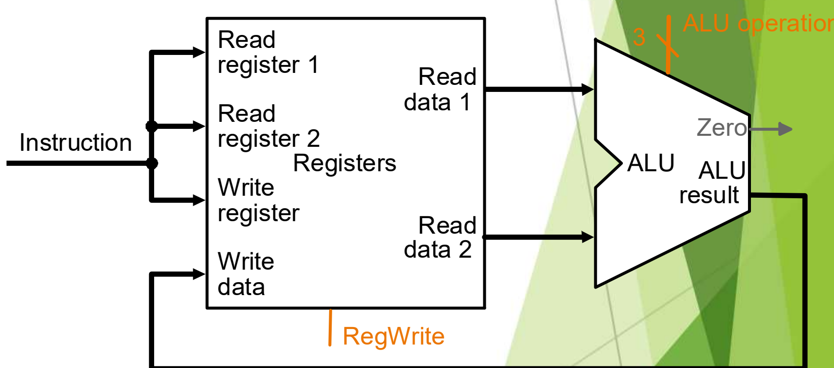
Datapath: R-Type Instruction



a. Registers



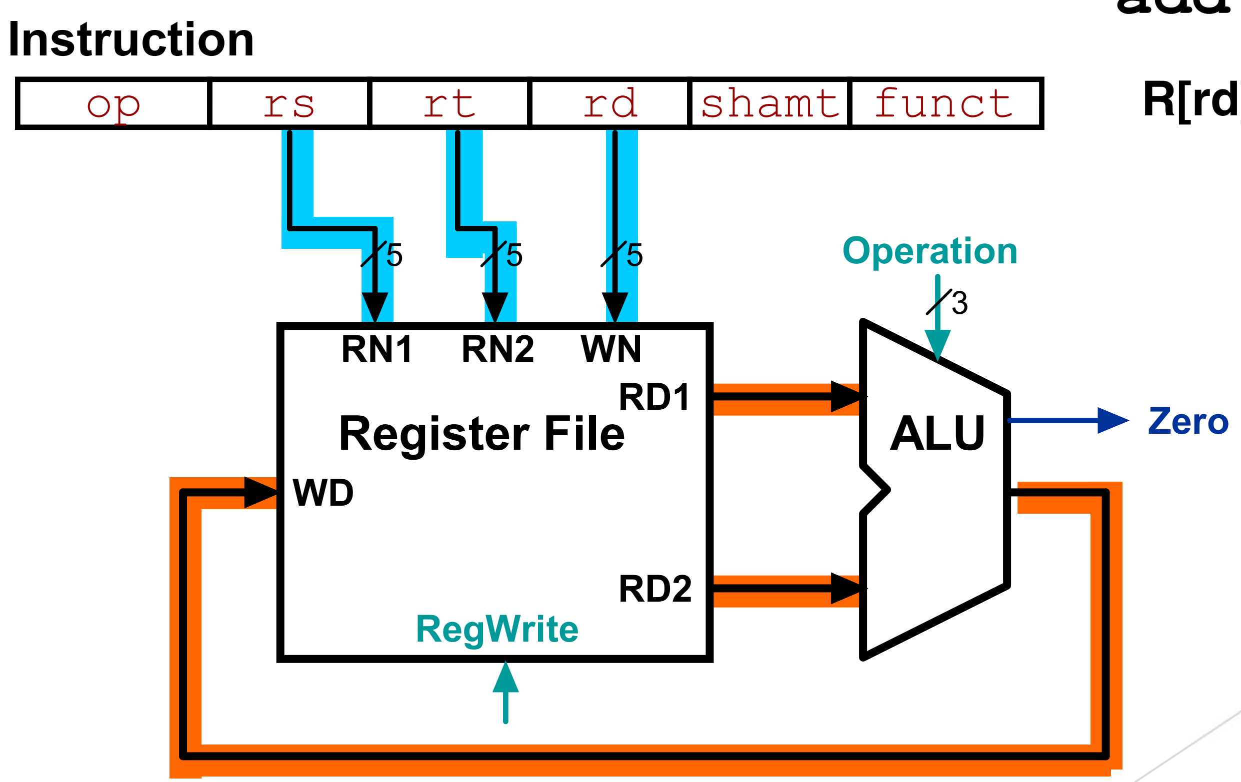
b. ALU



Two elements used to implement R-type instructions

Datapath

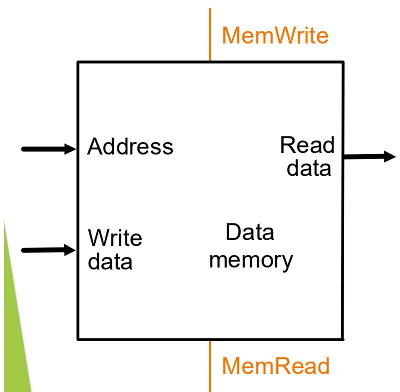
Animating the Datapath



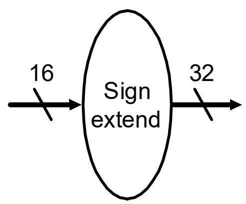
add rd, rs, rt

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

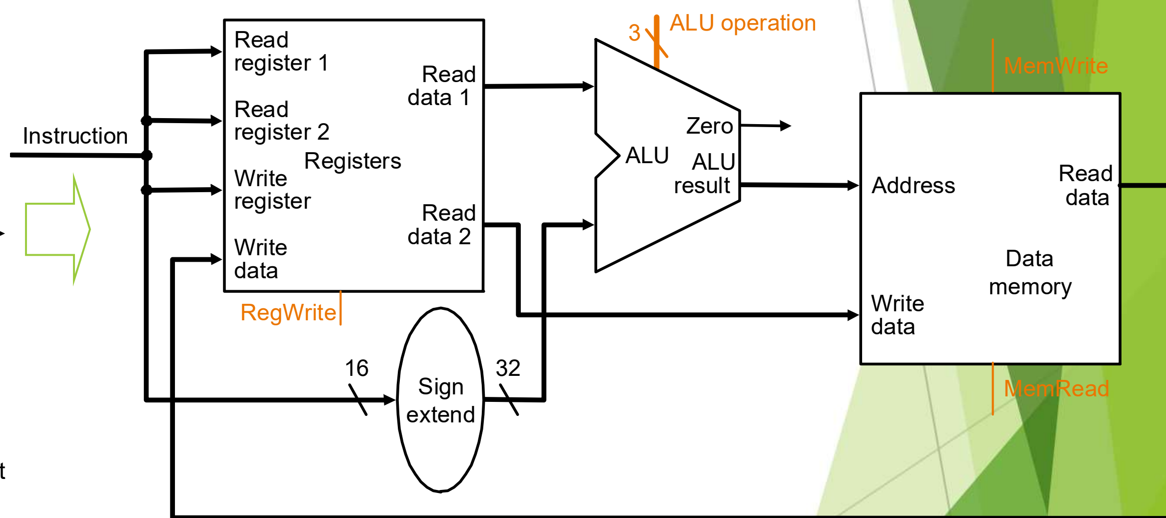
Datapath: Load/Store Instruction



a. Data memory unit



b. Sign-extension unit



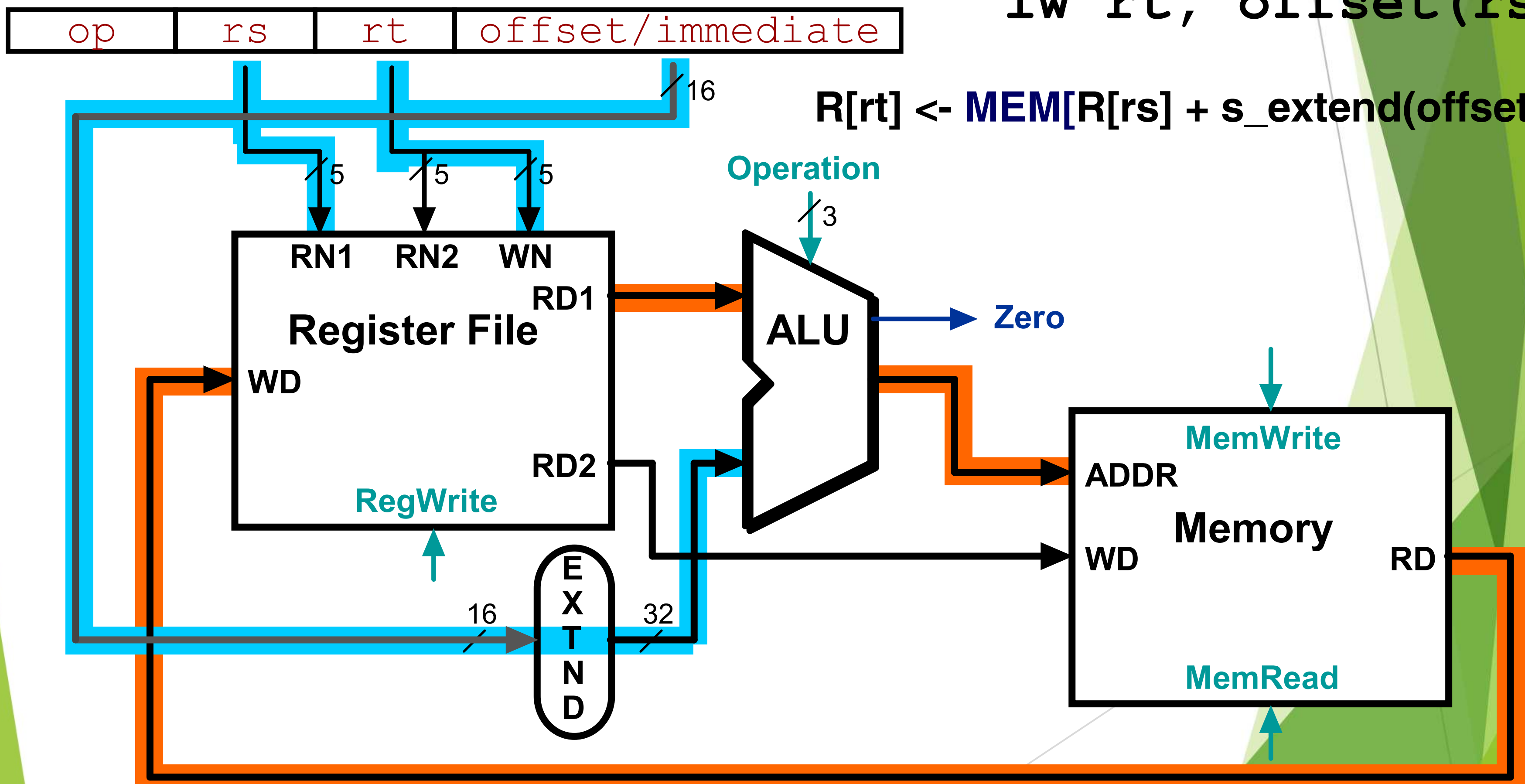
Two additional elements used
To implement load/stores

Datapath

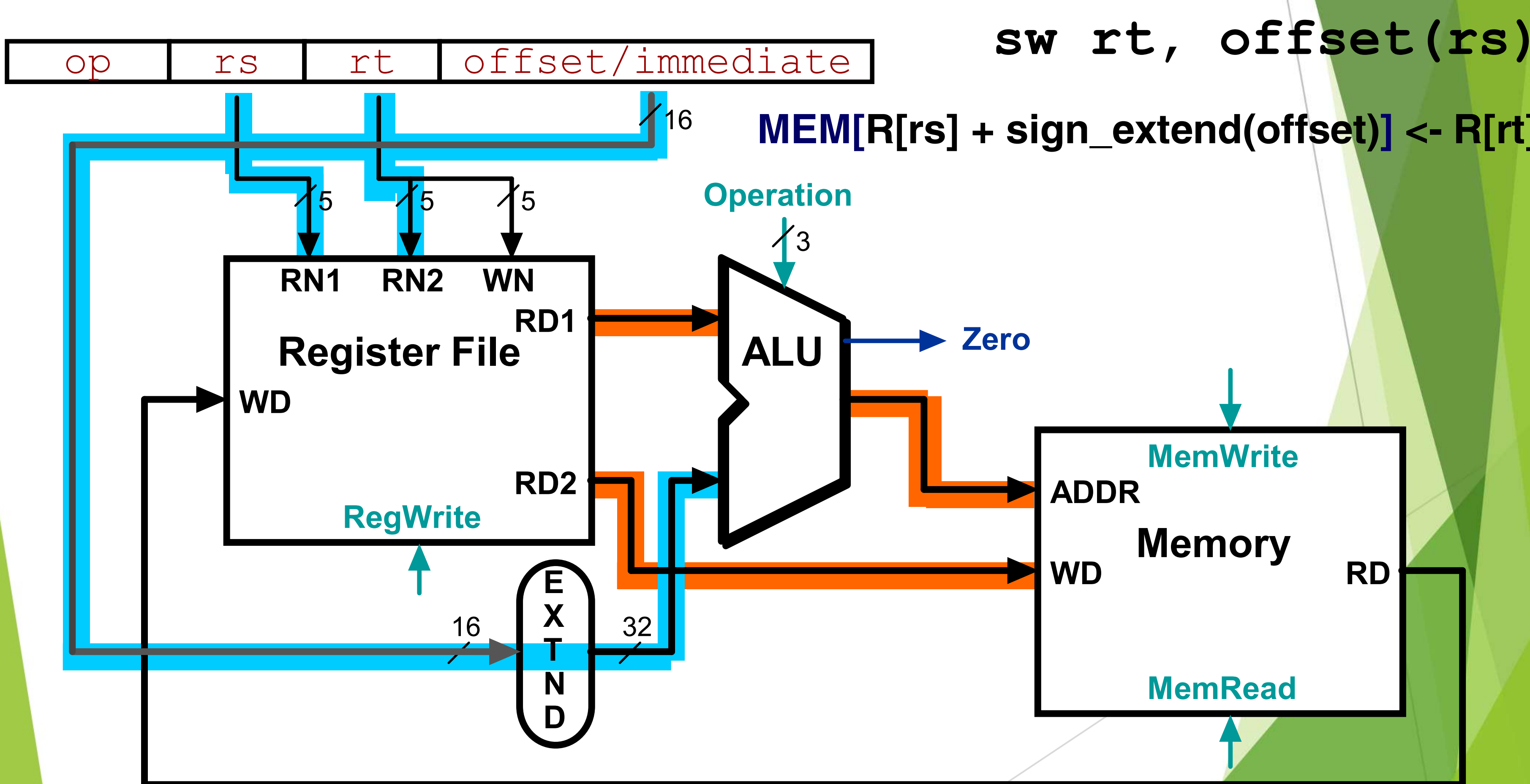
Animating the Datapath

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

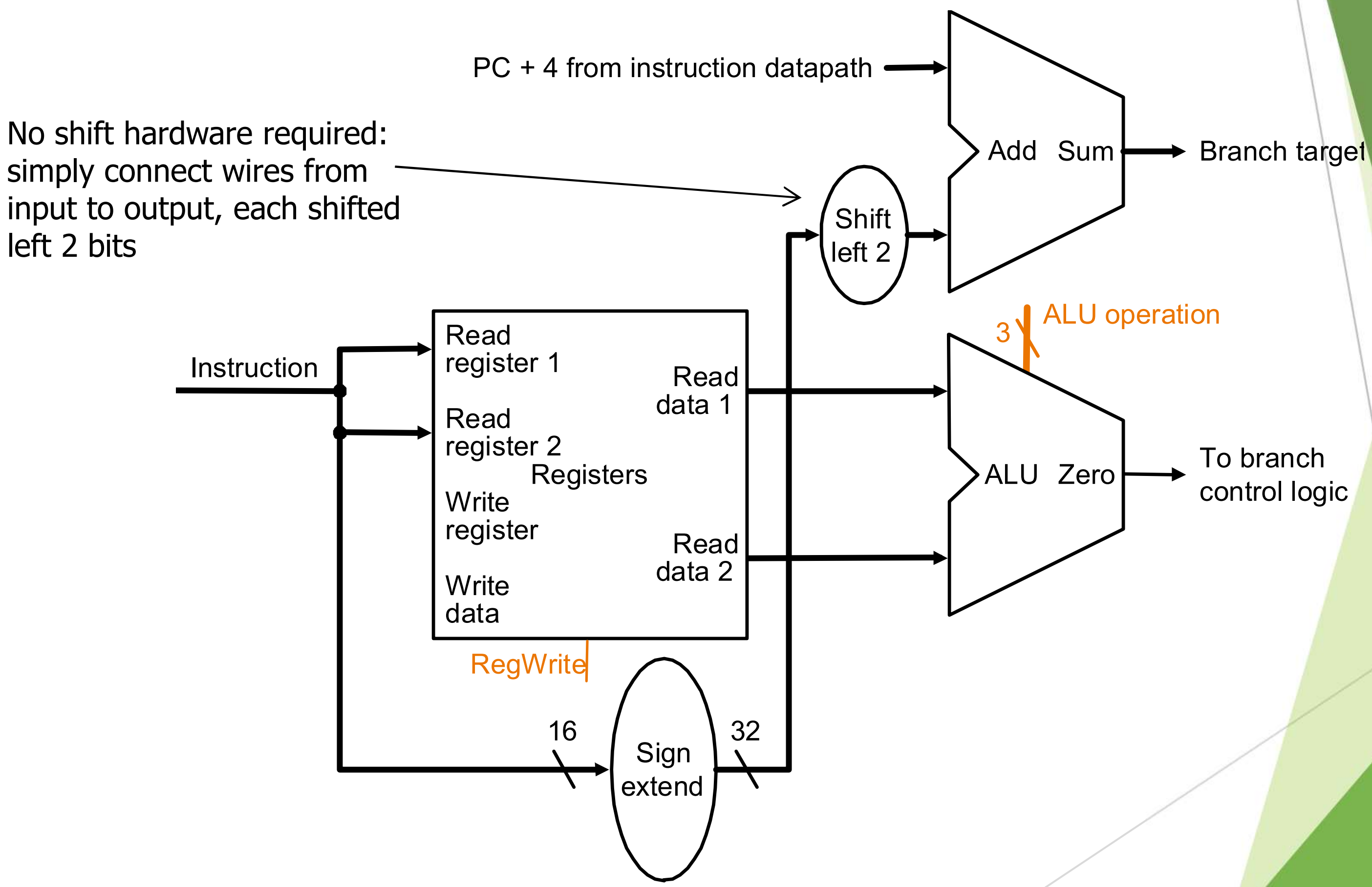
R[rt] <- MEM[R[rs] + s_extend(offset)];



Animating the Datapath

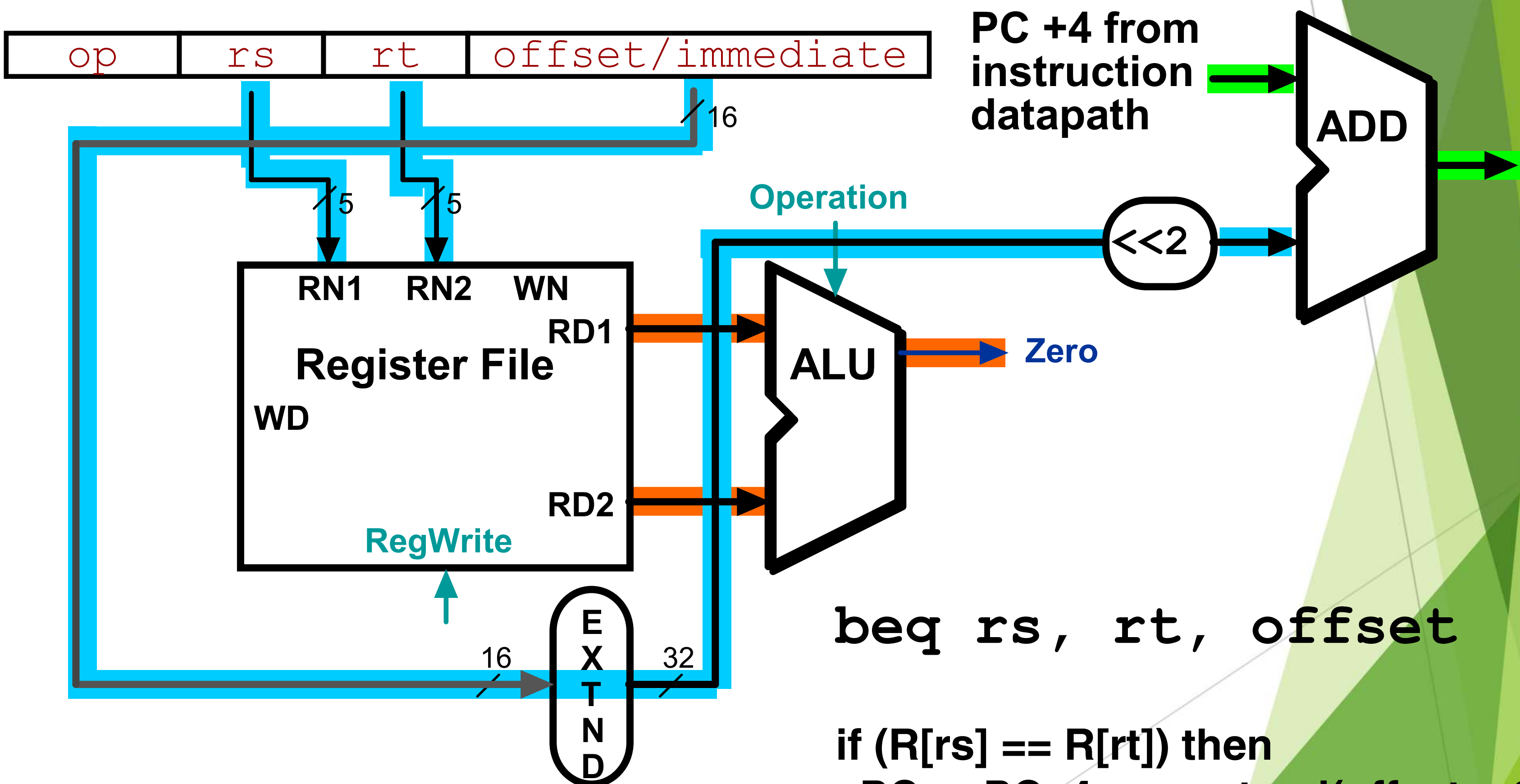


Datapath: Branch Instruction



Datapath

Animating the Datapath



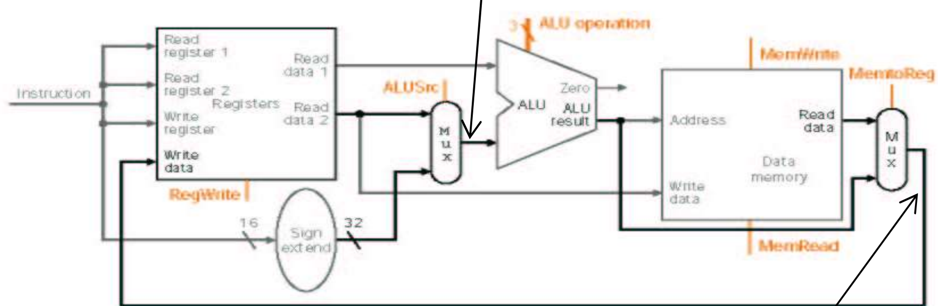
beq rs, rt, offset

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

MIPS Datapath I: Single-Cycle

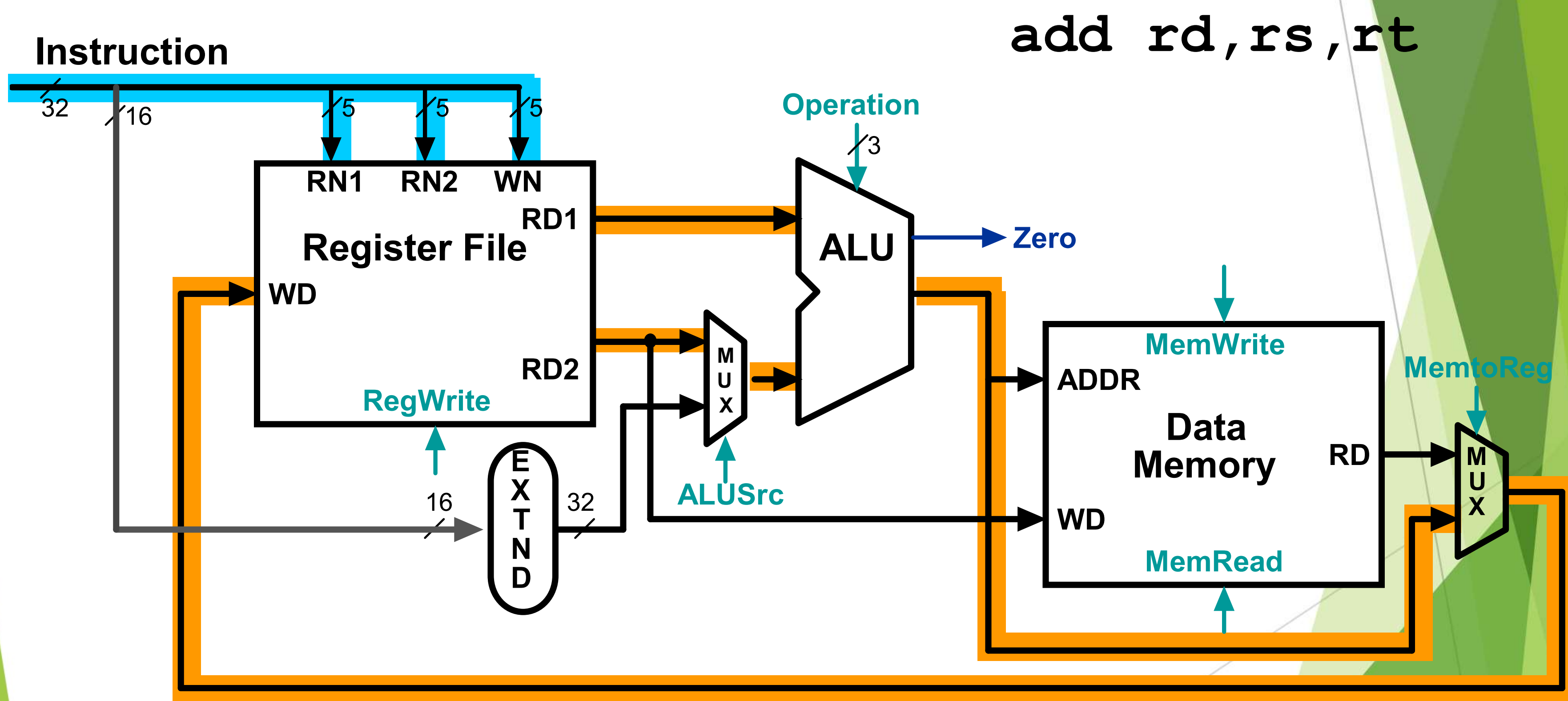
Combining the datapaths for R-type instructions and load/stores using two multiplexors

Input is either register (R-type) or sign-extended lower half of instruction (load/store)



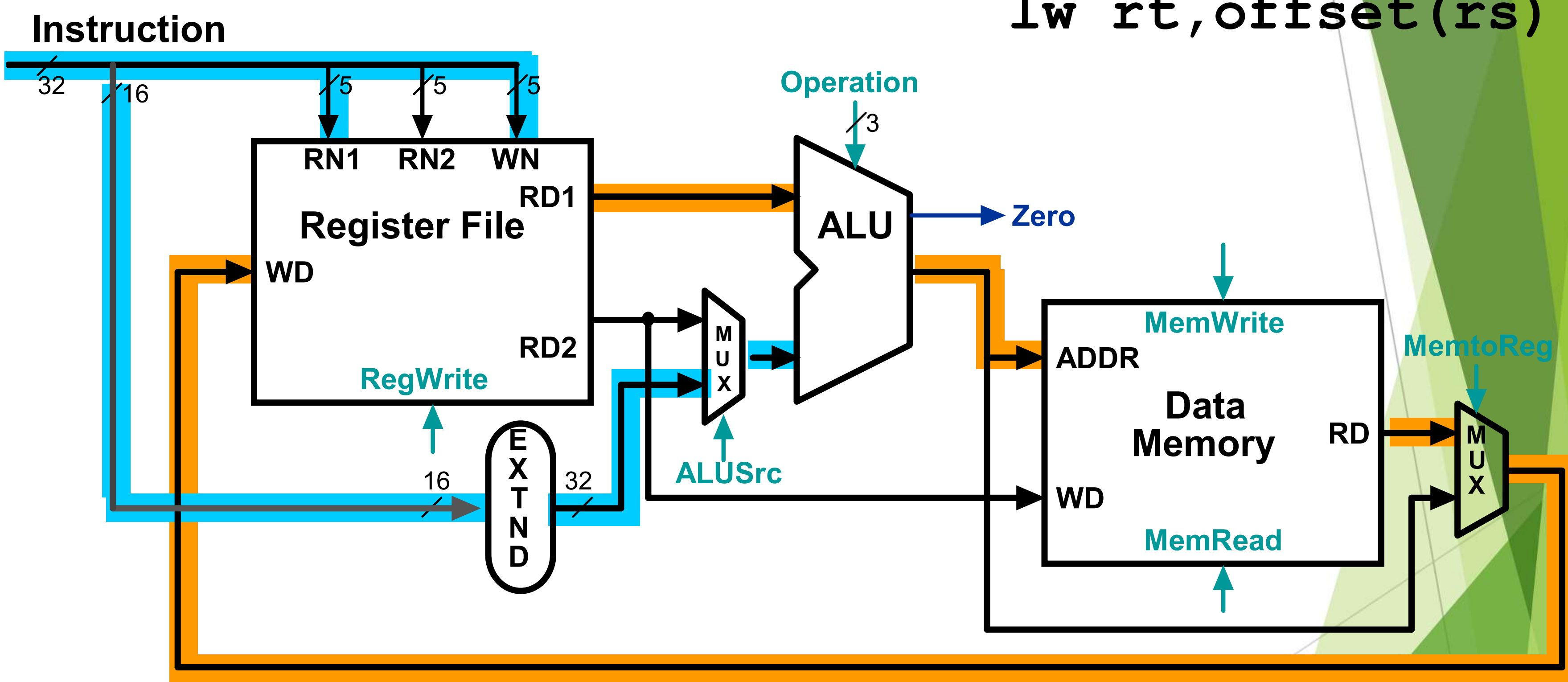
Data is either
from ALU (R-type)
or memory (load)

Animating the Datapath: R-type Instruction

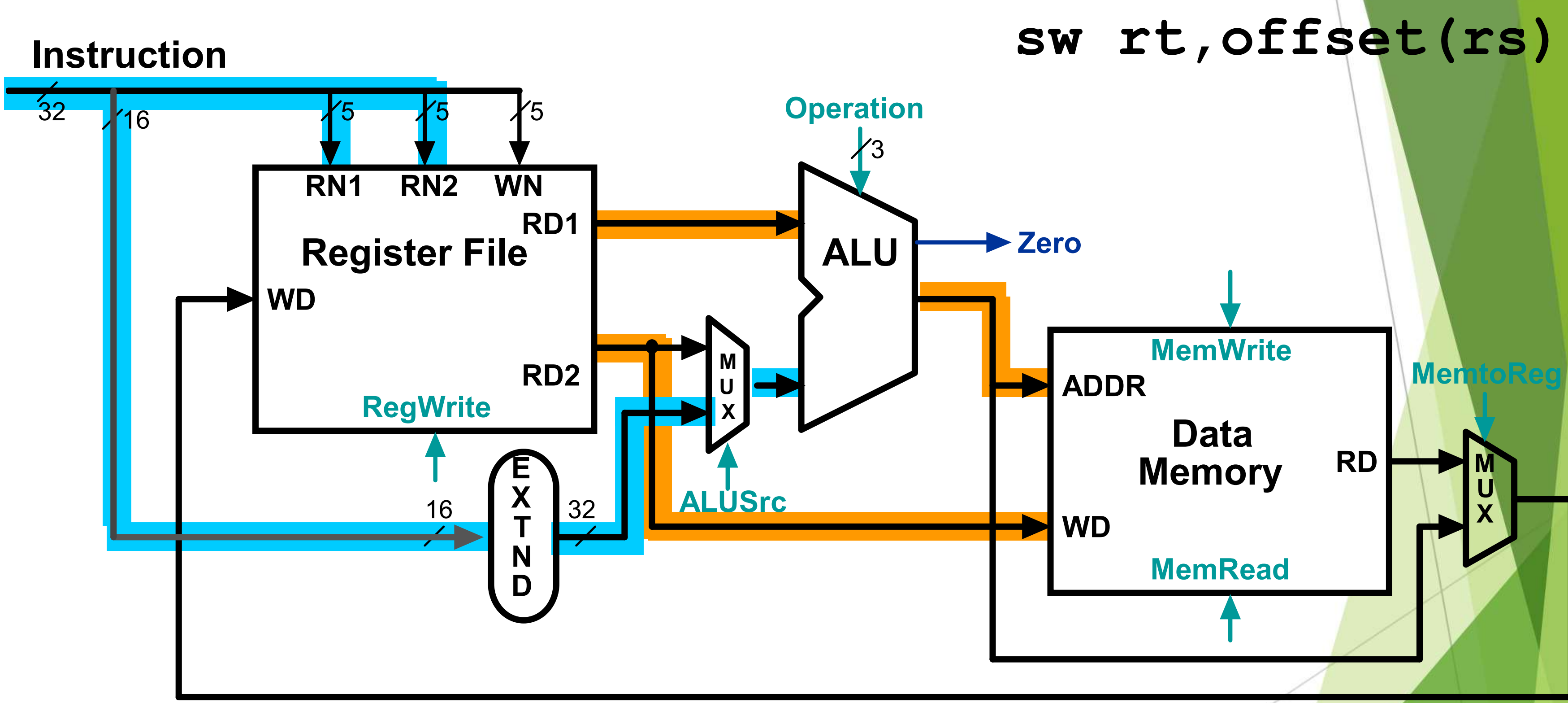


Animating the Datapath: Load Instruction

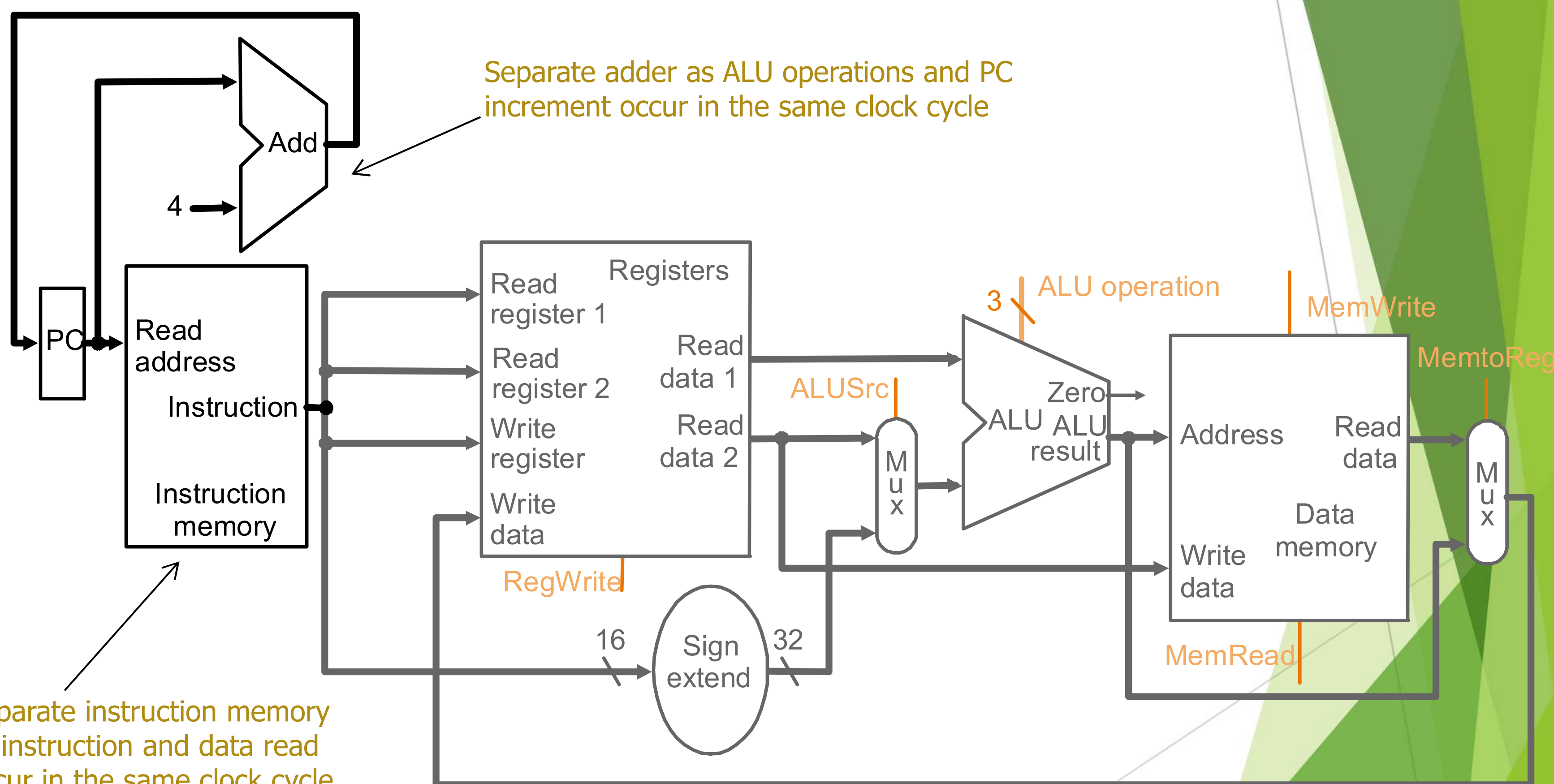
`lw rt,offset(rs)`



Animating the Datapath: Store Instruction



MIPS Datapath II: Single-Cycle

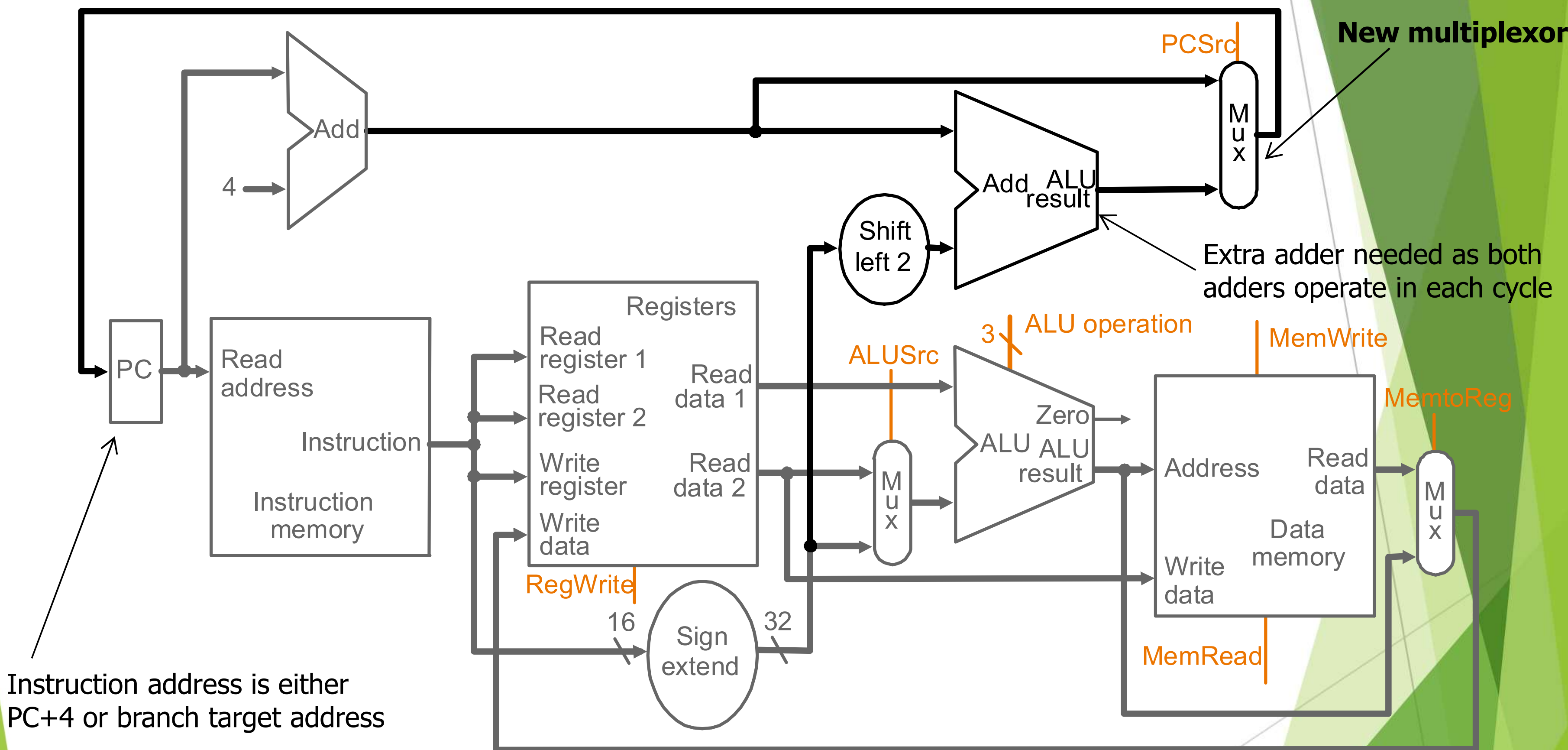


Separate adder as ALU operations and PC increment occur in the same clock cycle

Separate instruction memory as instruction and data read occur in the same clock cycle

Adding instruction fetch

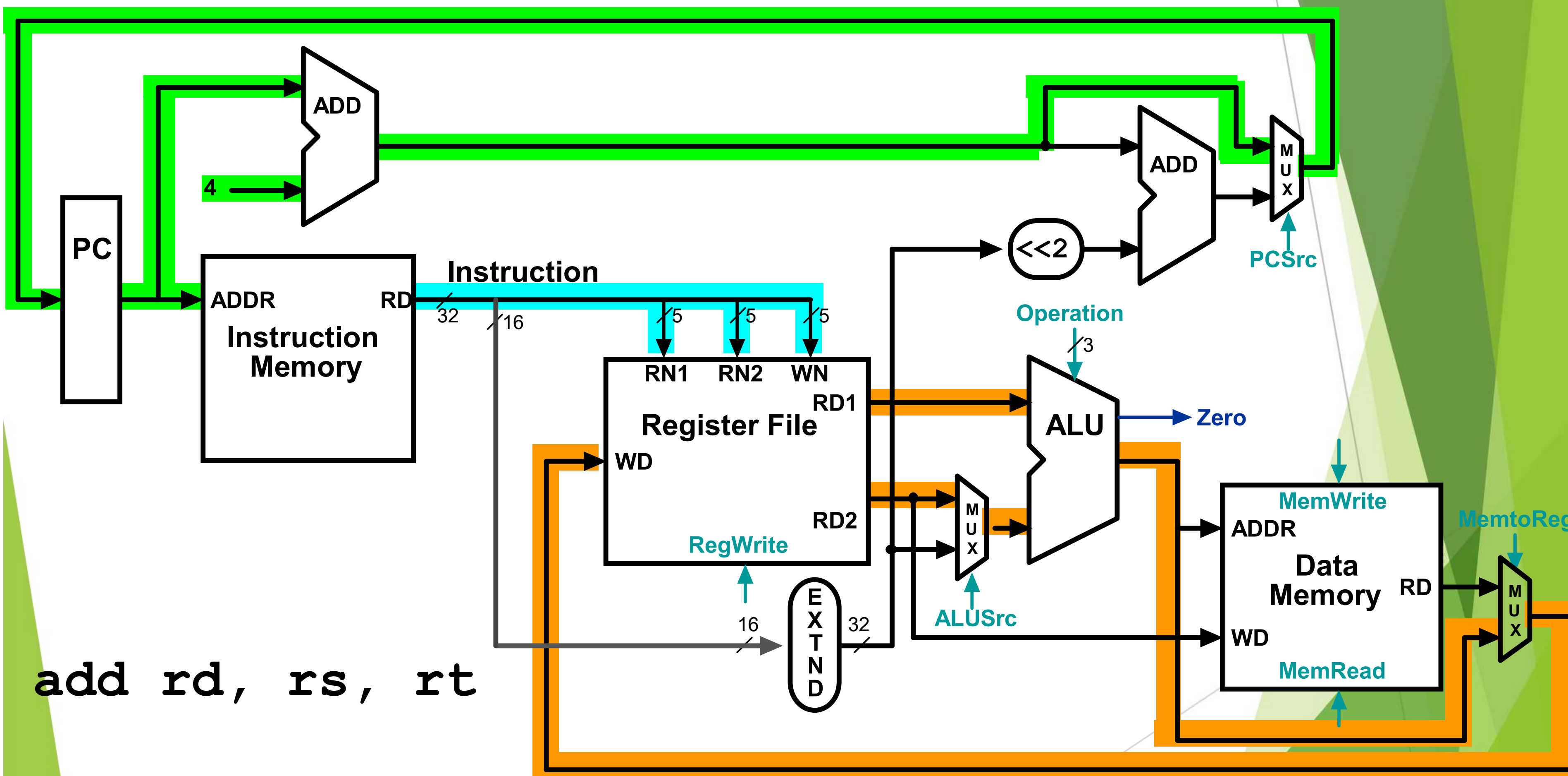
MIPS Datapath III: Single-Cycle



Adding branch capability and another multiplexor

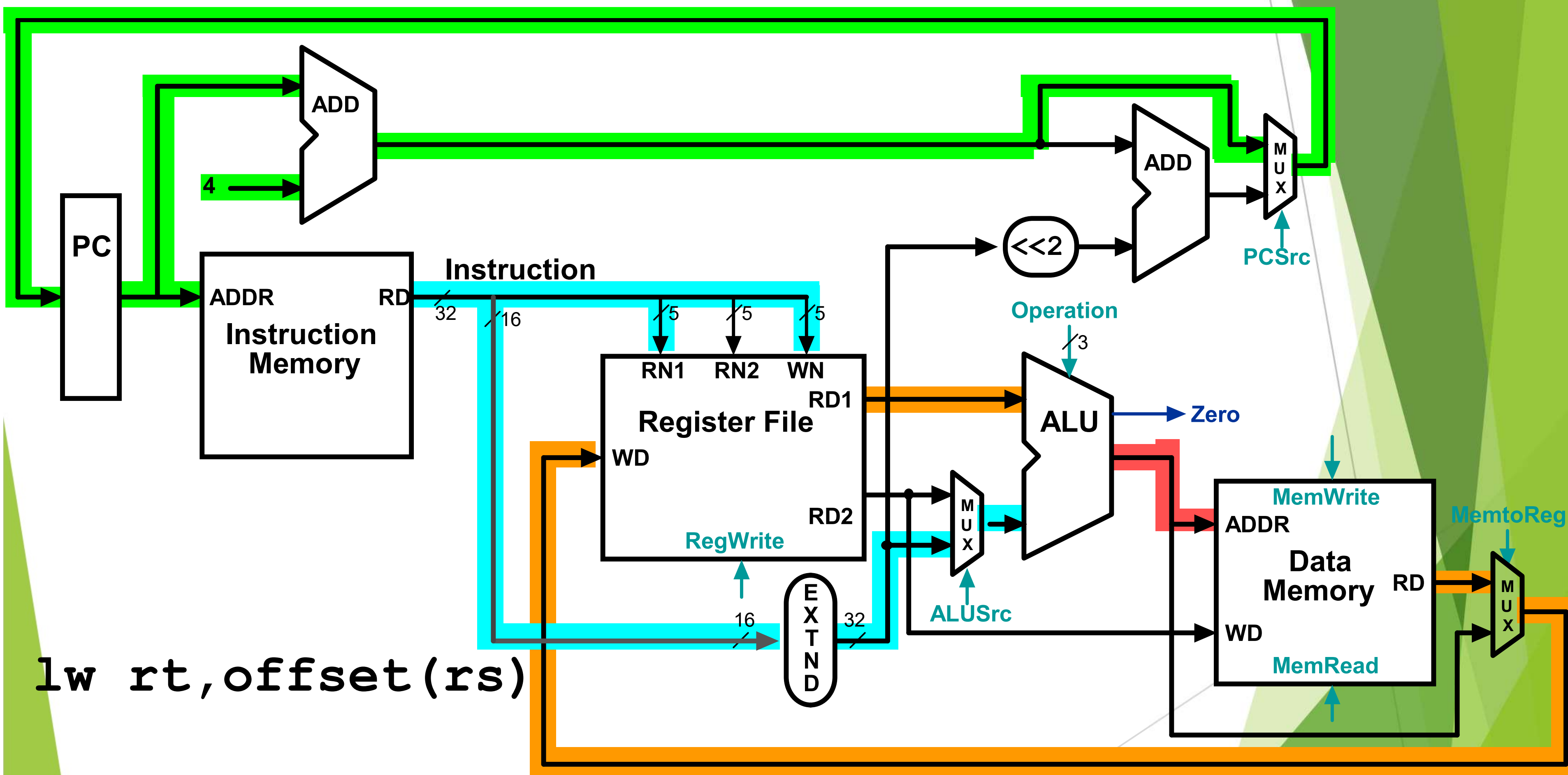
Important note: in a single-cycle implementation data cannot be stored during an instruction – it only moves through combinational logic
Question: is the MemRead signal really needed?! Think of RegWrite…!

Datapath Executing `add`



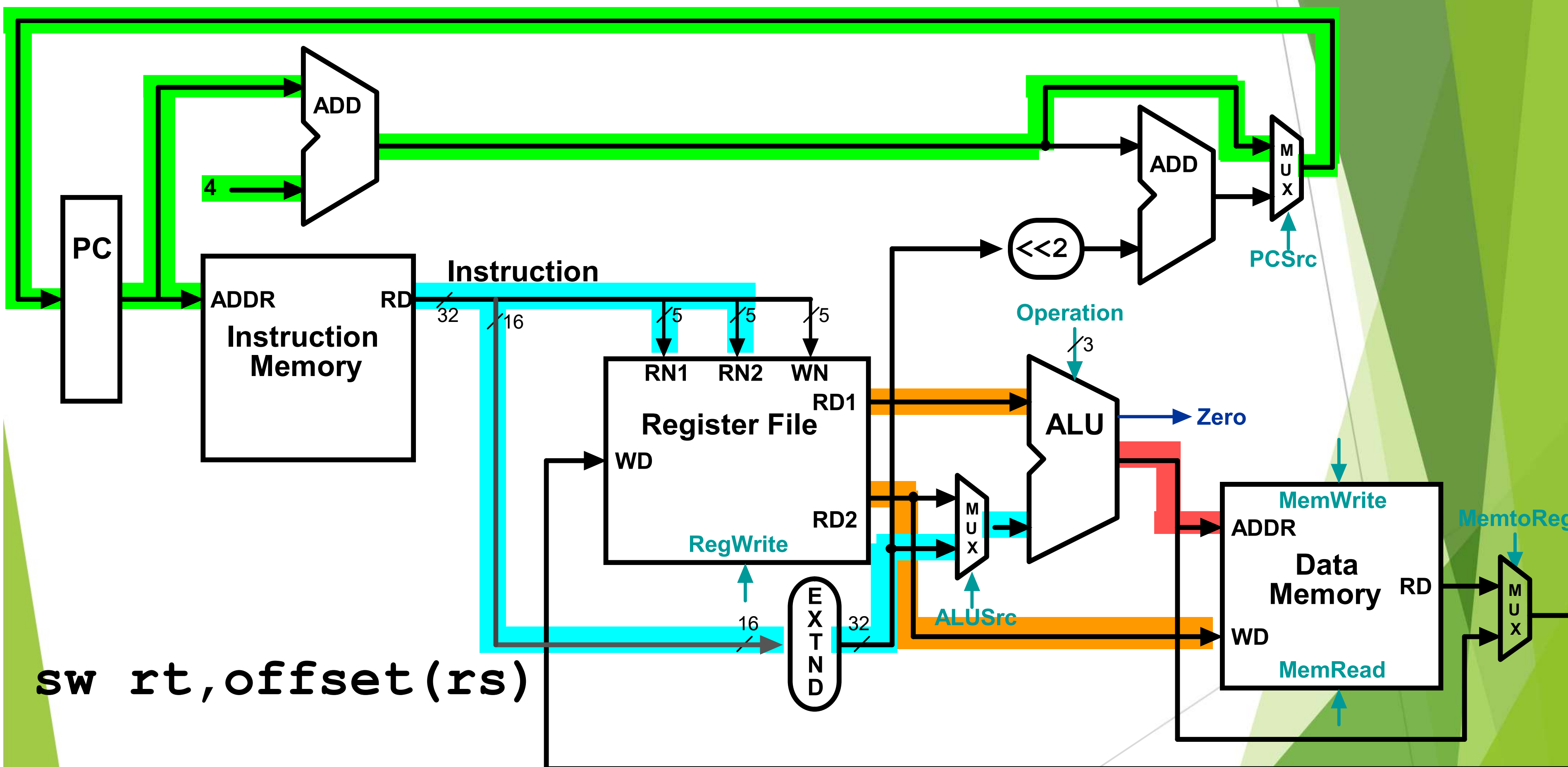
`add rd, rs, rt`

Datapath Executing `lw`



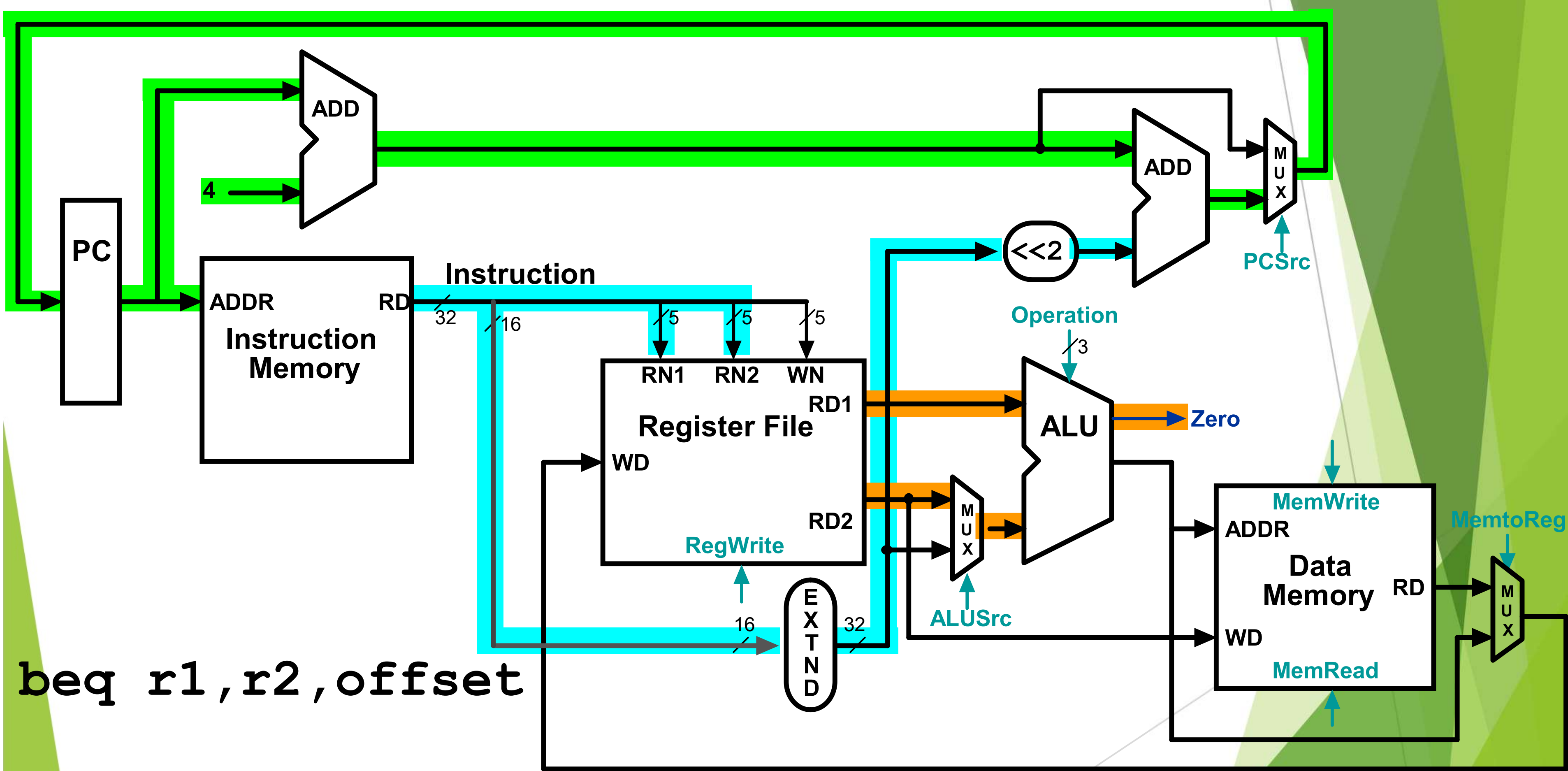
`lw rt,offset(rs)`

Datapath Executing `sw`



`sw rt,offset(rs)`

Datapath Executing beq



beq r1,r2,offset