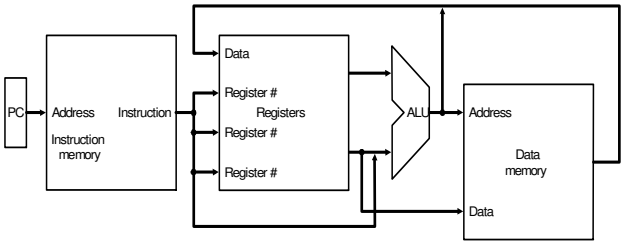
## The Processor: Datapath & Control

- Simplified implementation of MIPS with:
  - memory-reference: `lw, sw`
  - arithmetic-logical: `add, sub, and, or, slt`
  - control flow: `beq, j`
- Generic Implementation (Fetch + Execute):
  - use PC register to supply instruction address
  - read the instruction from memory at PC
  - read the registers specified by instruction
  - use instruction opcode to decide what to do
- All instructions (except one!) need to use an ALU after reading the registers. Why? E.g., memory-reference? arithmetic? control flow?



## Single Cycle Implementation Details

- Abstract / Simplified View:

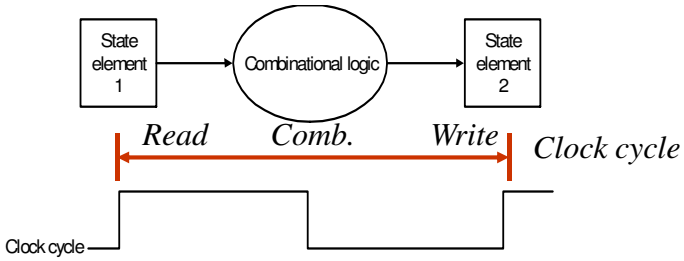



Two types of functional units:

- elements that operate on data values (combinational)
- elements that contain state (sequential)

## Our Implementation - Edge triggered

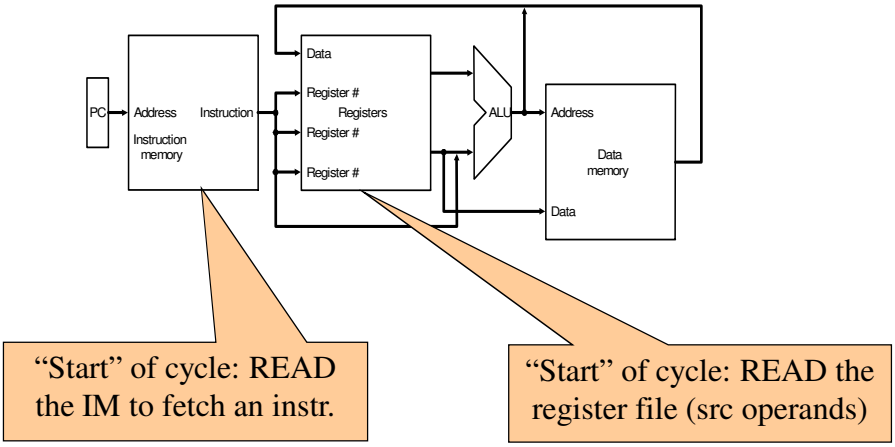
- **Typical execution:**
  - **read contents of some state elements**
  - **send values through some combinational logic**
  - **write results to one or more state elements**





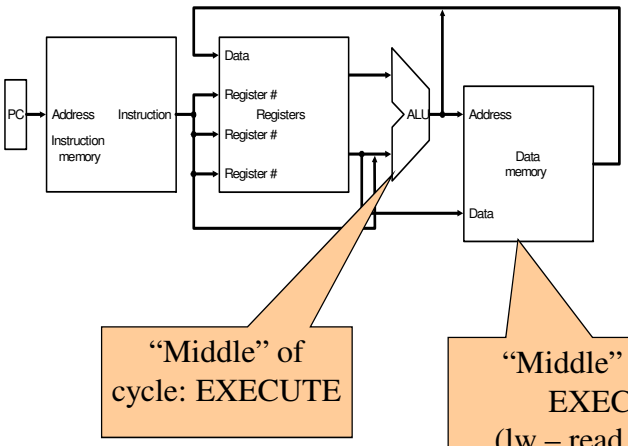

## A Single Cycle Implementation

- **Abstract / Simplified View:**



# A Single Cycle Implementation

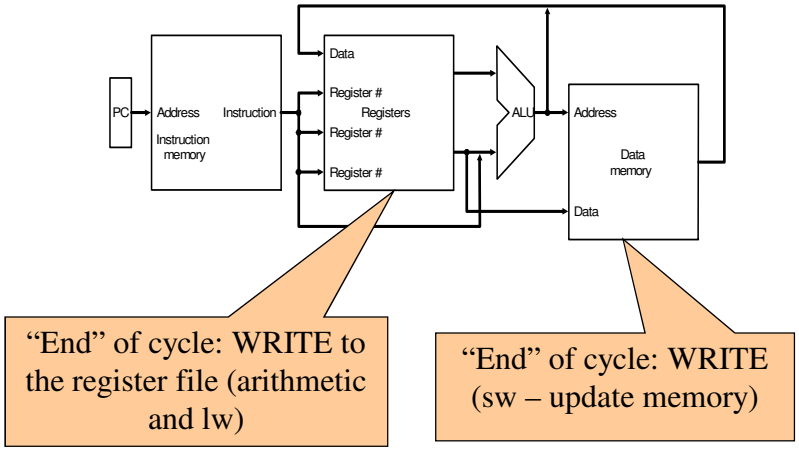
- **Abstract / Simplified View:**






# A Single Cycle Implementation

- **Abstract / Simplified View:**

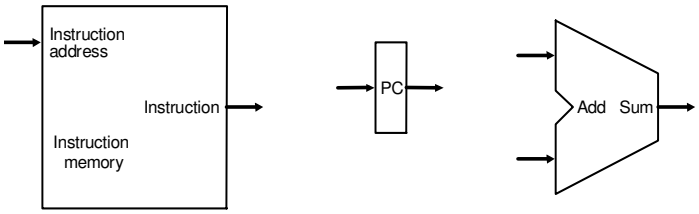


# Building a Data path: Start with Fetch

- **Components to execute each class of instruction**

**Fetch components**

- **Program counter - tracks address of instruction**
- **Instruction memory - memory from which we fetch an instruction to execute**
- **Adder - does increment of PC by 4, labeled as "add"**

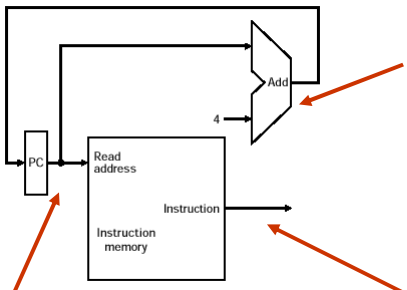



a. Instruction memory b. Program counter c. Adder



# Fetching Instructions

- **Fetch - get instruction from memory**




Increment PC by four after reading current instruction - by four since an instruction is 32 bits long

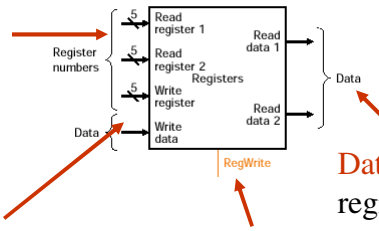
Instruction read from the memory - send to rest of the data path

Send address from PC to the instruction memory to read the instruction at IM[PC]

## Accessing Registers

- **After fetching, we need to get source operands for the instruction - read the register file**
- **Register file - collection of registers that can be read or written by specifying register number**

Address of the register to read - we need to read two registers, so there are two "read ports"




Data read from the register file

For a write, we need to send only one destination register (number) and the data (one "write port")

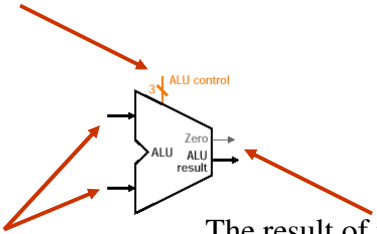
Whether this access is a read or a write operation



## Executing the Instruction

- **For an arithmetic instruction, once we have the operands, we need to execute it with the ALU**

What kind of operation the ALU is going to do for an instruction



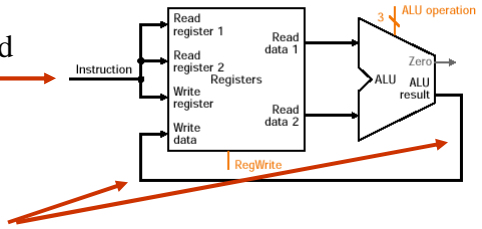

Two source operand inputs (e.g., read from the register file for R-type)

The result of the operation done on the ALU
Includes "zero" detection - whether the output is zero

## R-type Instructions (Arithmetic) Data Path

- From register file and ALU, we can create the data path for arithmetic instructions
- Connect read outputs from register file to ALU
- Connect ALU output to register file write data input

Register numbers come from the fixed fields in the instruction




Output from the ALU fed back to the register file write data input



## How 'bout Load and Store?

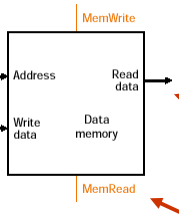
- Recall load and store:
  - lw $t1,0($t2)
  - sw $t1,12($t2)
- Effective address - the address from which we will load or store a value
  - Formed by adding offset_value + $basereg
- offset_value is a 16-bit immediate (constant)
  - we add on 32-bit values (using the ALU!)
  - need to "sign extend" the value

## Elements for Load and Store

- **Data memory element - holds program data**

Access to the memory is a write operation - use address to locate place in memory and write the new value in that place

Address of the item to read/write from/to memory for a load/store




Data read from the memory

Data value to write to memory for a store

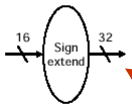
Access to the memory is a read operation - use address to locate value and send it to the output



## Elements for Load and Store

- **Sign extend - replicate the sign (msb) of the immediate offset in the top 16 bits of the 32 bit input to the adder (for forming the effective address)**

16-bit input from the immediate field of the load or store instruction

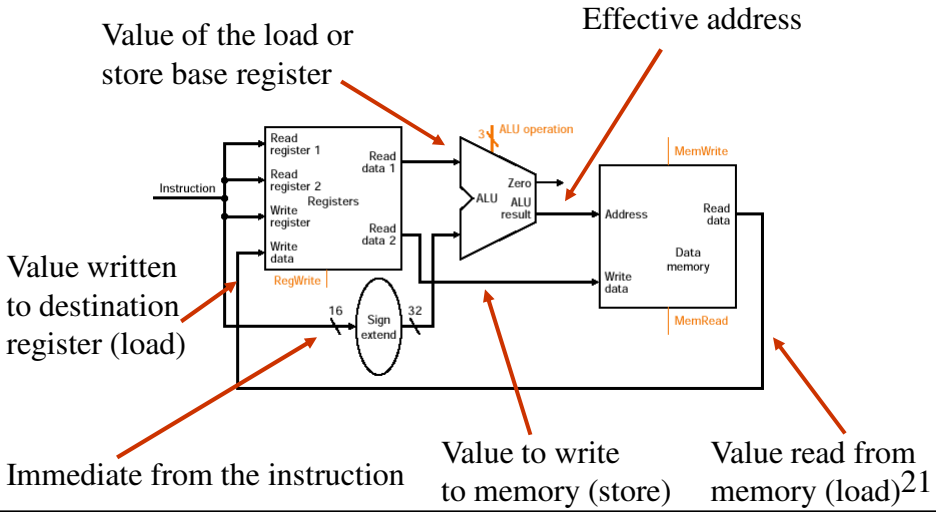



*e.g.,16-bit offset -104 is 0xFF98*
*sign_ext(0xFF98) = 0xFFFFFF98*

32-bit sign extended output (going to the ALU for the E.A. addition)

## Data path for Load and Store

- **Combine the register file, sign extend, ALU, and the data memory**






## OK, now, what about branches?

- **Recall that the branches are:**
  - **beq $t1,$t2,label  Branch to "label" if $t1 == $t2**
  - **bne $t1,$t2,label  Branch to "label" if $t1 != $t2**

- **So, we have:**
  - **2 register source operands**
  - **a 16-bit offset (PC-relative)**
  - **ALU to do comparison (remember zero check?)**

- **When branch taken, PC is set to (PC+4) + (offset<<2)**

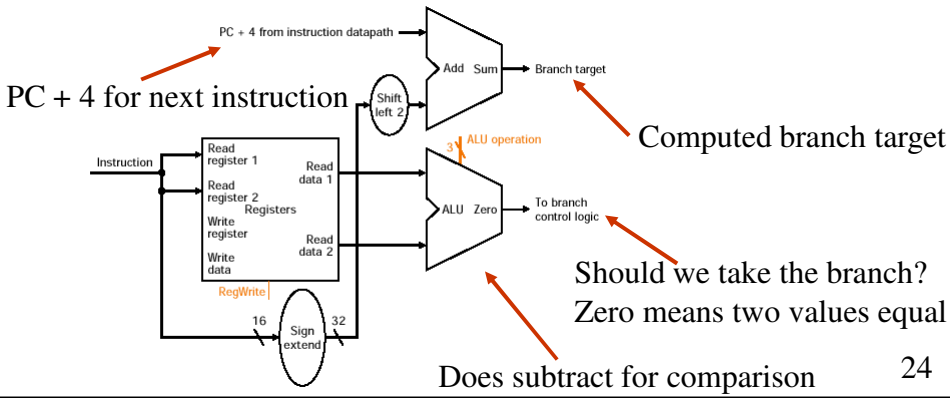
## More about branches

- For PC-relative branches on MIPS,
  - the "base value" is actually the PC + 4
  - the offset is also shifted left by 2 (word aligned)
- PC + 4 value is available after we increment the PC to fetch the next instruction
- Wait a minute….
  - We need to compute the target address
  - AND
  - use the ALU to compare $t1 and $t2
- What do we do????



## Branch Elements

- A second adder - does target address calculation
- Shifter - shift offset by 2
- A way to set the PC to the branch target address if the branch is actually taken ("taken vs. not taken")



## Jumps

- Recall that jump is:
  - j label
  - label is a 26 bit value
- Simply replaces PC with
  - 26 bit immediate shifted left by 2
  - PC=PC[31..28] . (Imm26<<2)