SINGLE CYCLE VS MULTI CYCLE CPU ARCHITECTURE

A **single cycle** cpu executes each instruction in one cycle. in other words, one cycle is needed to execute any instruction. in other words, our cpi is 1.

Each cycle requires some constant amount of time. This means we will spend the same amount of time to execute every instruction [one cycle], regardless of how complex our instructions may be. to ensure that our processor operates correctly, our slowest instruction must be able to complete execution correctly in one clock tick. This is the big disadvantage of single cycle cpu's: the machine must operate at the speed of the slowest instruction. The big advantage of single cycle cpu's is that they are easy to implement.

As its name implies, the multiple cycle cpu requires multiple cycles to execute a single instruction. This means that our cpi will be greater than 1. The big advantage of the multi-cycle design is that we can use more or less cycles to execute each instruction, depending on the complexity of the instruction. for example, we can take five cycles to execute a load instruction, but we can take just three cycles to execute a branch instruction. The big disadvantage of the multi-cycle design is increased complexity. Control is now a finite state machine - before it was just combinational logic.

Another important difference between the single-cycle design and the multi-cycle design is the cycle time. In the single cycle processor, the cycle time was determined by the slowest instruction. In the multi-cycle design, the cycle time is determined by the slowest functional unit [memory, registers, alu]. This greatly reduces our cycle time.

A single instruction enters the CPU at the Fetch stage and the PC is incremented in one clock cycle. In the next clock cycle, the instruction moves to the Decode stage. In the third clock cycle, the instruction moves to the Access stage and the operands are loaded. In the last two stages, the instruction is executed and the result is stored. In a five stage pipeline a single instruction will take 5 clock cycles to pass through the pipeline. Since the pipeline stages operate independently, a new instruction may enter the Fetch stage as soon as the add instruction has moved to the Decode stage. Under ideal circumstances, a pipelined processor can produce a result on every clock cycle. **Thus, the peak MIPS (Millions of Instructions Per Second) rating of the CPU equals the clock speed in Mhz.** A pipelined CPU achieves maximum throughput only when all stages of the pipeline are filled with instructions which can be processed independently. Performance decreases when gaps or holes appear in the pipeline. A hole is an empty pipeline stage which is not processing an instruction due to hazards in pipeline-ing e.g data hazard , resource hazard or control hazard.

Single cycle CPU micro-architecture

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} * \frac{\text{Cycles}}{\text{Instruction}} * \frac{\text{Time}}{\text{Cycle}}$$

Let we have to design a cpu whose instruction length is of 16 bits size. Instructions have two register operands/ addresses or single memory operand/ address. We have reserved 4 bits for opcode. And for register addresses we have 5 bits for each register address. What is needed to execute instruction... e.g data path and control path (register file and Alu (is just combinational logic) are main parts).

The datapath is the unit that contains all the registers and the functional units. It is where all data computations take place. *The datapath should be constructed in a way such that it is possible to perform all actions necessary to compute the data results of instruction execution.* For example, to implement the add instruction the datapath should provide us with the capability of reading the two registers that we want to add, it should have a functional unit that performs the addition and then allow us to write the result back into the appropriate register.

The control handles the actions that take place in the data path. Eventually the control unit is a finite-state machine or decoder that implements the fundamental execution loop: (1) fetch instruction, (2) decode, (3) read source operands, (4) perform operation, (5) store result, (6) determine which instruction to execute next. It does so by instructing the datapath to perform all appropriate actions.

CPU registers can be accommodated in this architecture???

As there are 5 bits per register address, 32 registers can be accommodated.....and size of each register should be of same size as of word size of main memory e.g of 16 bit size...

How many main memory locations can be addressed??

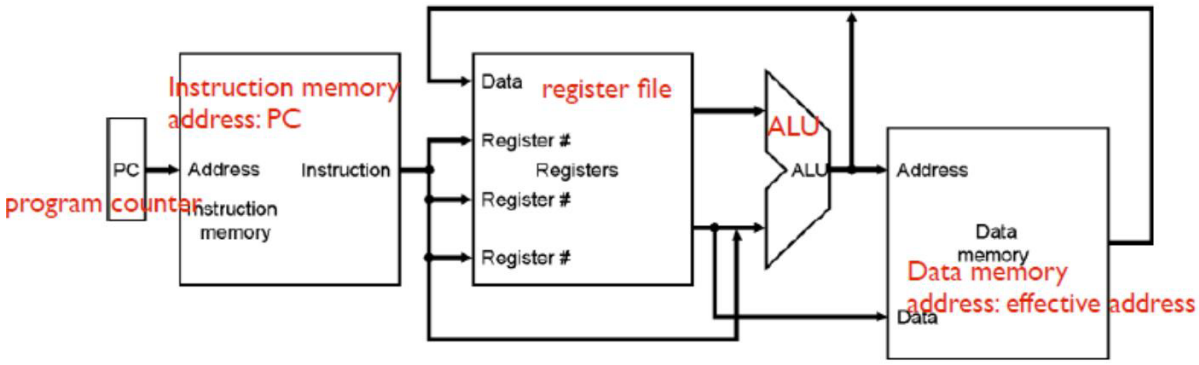
2 power 12 memory locations can be addressed as address part of instruction is of 12 bits when there is only memory operand/ address.

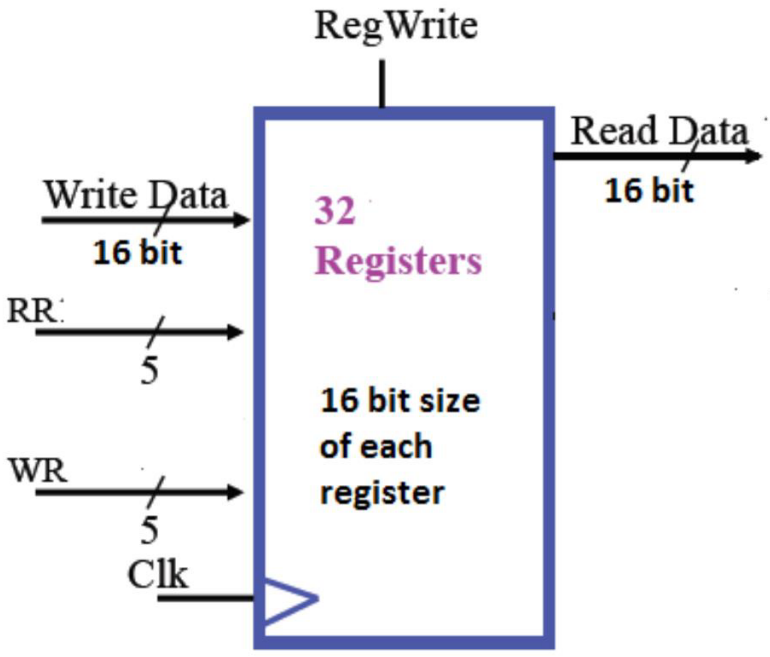
Register file and how it is implemented, how cpu hardware works when instruction is executed..e.g what will be on wires , address wires, control wires....etc.....???????

Register file (registers and combinational logic or finite state machine)

Although called a "file", a register file is not related to disk files. A register file is a small set of high-speed storage cells inside the CPU. There are special-purpose registers such as the IR and PC, and also general-purpose registers for storing operands of instructions such as add, sub, mul, etc. **A CPU register can generally be accessed in a single clock cycle, whereas main memory may require dozens of CPU clock cycles to read or write**.

Since there are very few registers compared to memory cells, registers also require far fewer bits to specify which register to use. This in turn allows for smaller instruction codes. For example, the processor we are designing has 32 general-purpose registers, so it takes 5 bits to specify which one to use. In contrast, it has a 2K memory capacity of 16 bit width, so it takes 12 bits to specify which memory array/ cell to use (word addressing is used). Below is diagram for single cycle cpu architecture and register file respectively.





Multi-cycle datapath

Multi-cycle implementaion: break up instructions into separate steps

Each step takes a single clock cycle

Each functional unit can be used more than once in an instruction,

as long as it is used in different clock cycles

Reduces amount of hardware needed

Reduces average instruction time

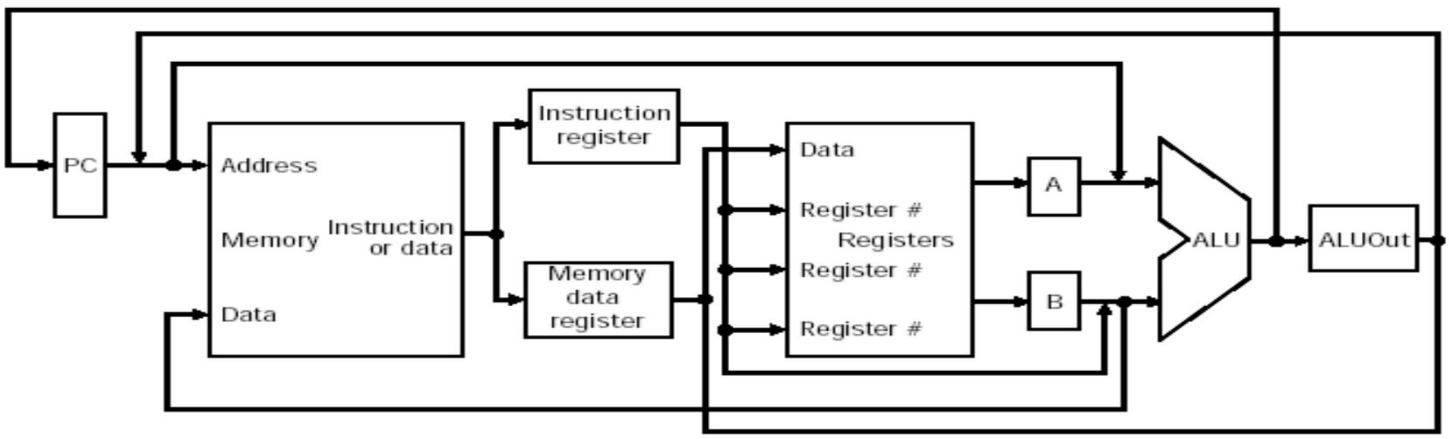
Differences with single-cycle

Single memory for instructions and data

Single ALU (no separate adders for PC or branch calculation)

Extra registers added after major functional units to hold results between clock cycles

Fig. 5.30



Note that data needed in a later instruction must be in one of the
programmer-visible registers or memory

Assume each clock cycle includes at most one of:
Memory access
Register file access (2 reads OR 1 write)
ALU operation

Any data produced from 1 of these 3 functional units must be stored between cycles
Instruction register: contains current instruction
Memory data register: data from main memory
Why 2 separate registers? Because both values are needed simultaneously

Register output A, B
2 operand values read from register file

ALUOut
Output from ALU
Why is this needed? Because we are combining adders into the ALU,
so we need to select where the output goes (register file or memory)

All these registers except IR hold data only between consecutive clock cycles,
so don't need write control signal

What else do we need?

Because functional units are used for multiple purposes:
More MUXes
More inputs for existing MUXes

Multi-cycle datapath

MUX example 1:

One memory is used for instructions and data, so we need a MUX to select between:

- PC (instruction)
- ALUout (data)
- for address to access in memory

Where else? (Hint: Consider ALU)

MUX example 2:

One ALU is used to perform all arithmetic and logic operations, so we need a MUX to select first operand between

- PC
- Data register A

Also, for second operand:

- Data register B
- Sign-extended immediate
- Sign-extended/ shifted immediate (offset for branch)
- Constant 4 (incrementing PC)

Multi-cycle datapath

Datapath with MUXes for selection:

4
2
1
5
3

MUX 1: select between PC and ALUOut for memory address

Fig. 5.31

MUX 2: select between $rt and $rd for destination (write) register address

MUX 3: select between ALUOut and memory data for write data input to register file

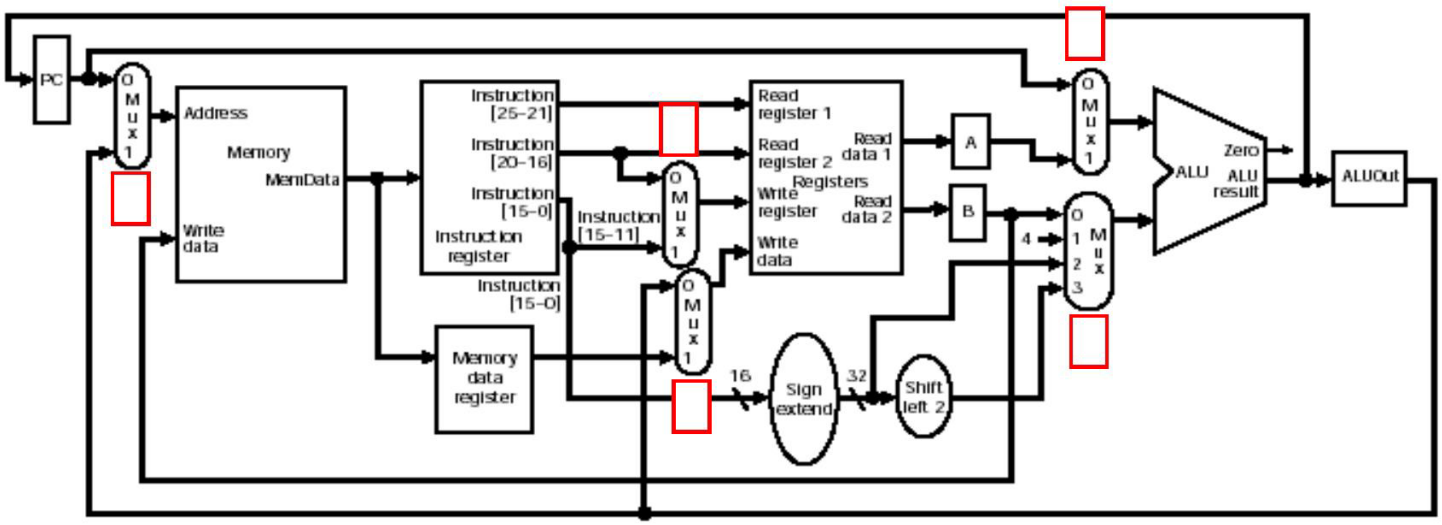
MUX 4: select between PC and register data A for first operand input to ALU

MUX 5: select between
register data B

constant 4

sign-extended immediate

sign-extended, shifted immediate
for second operand input to ALU



Multi-cycle datapath

Control signals needed to select inputs, outputs

Need write control:

Programmer-visible units

PC, memory, register file

IR: needs to hold instruction until end of execution

Need read control:

memory

ALU Control: can use same control as single-cycle

MUXes: single or double control lines (depending on 2 or 4 inputs)

Multi-cycle datapath: control signals

New control signals

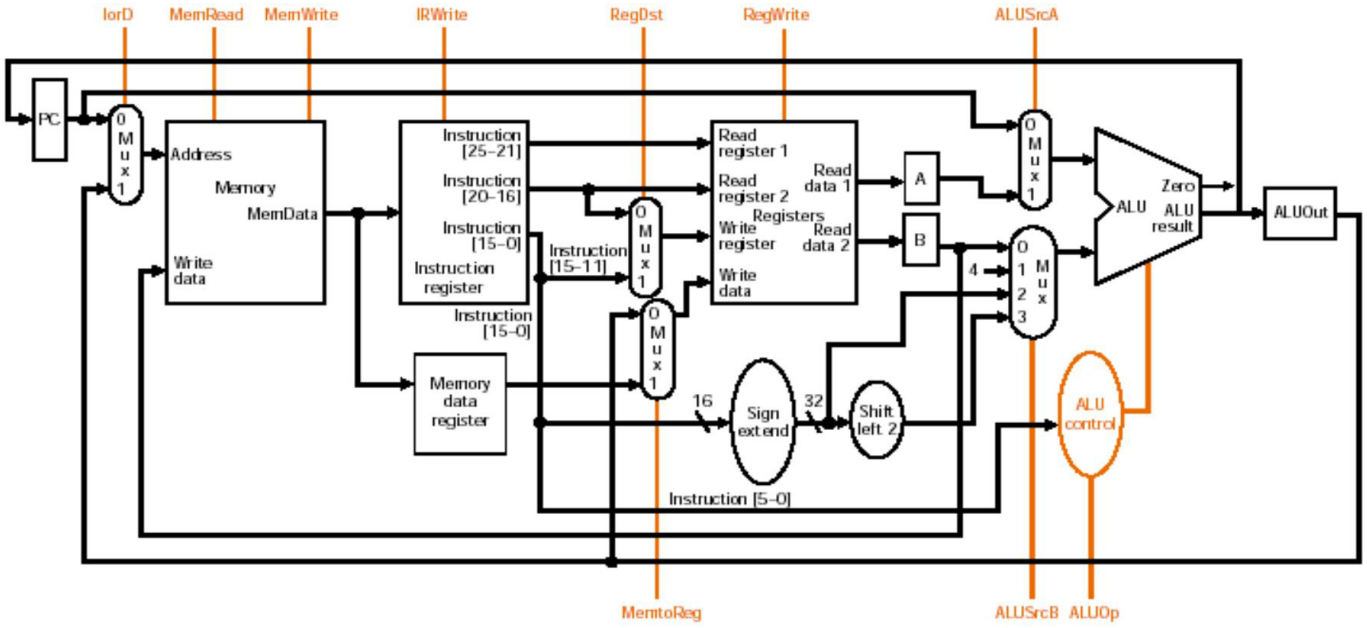
Fig. 5.32

IorD: selects PC (instruction) or ALUOut (data) for memory address

IRWrite: updates IR from memory (when?)

ALUSrcA: control to select PC or reg A (read data 1 from register file)
output is first operand for ALU

ALUSrcB: control to select second operand for ALU among 4 inputs:
0: reg B (read data 2 from register file)
1: constant 4
2: sign-extended immediate from instruction
3: above value shifted left by 2



Multi-cycle datapath: control signals

What else is needed? Branches and jumps

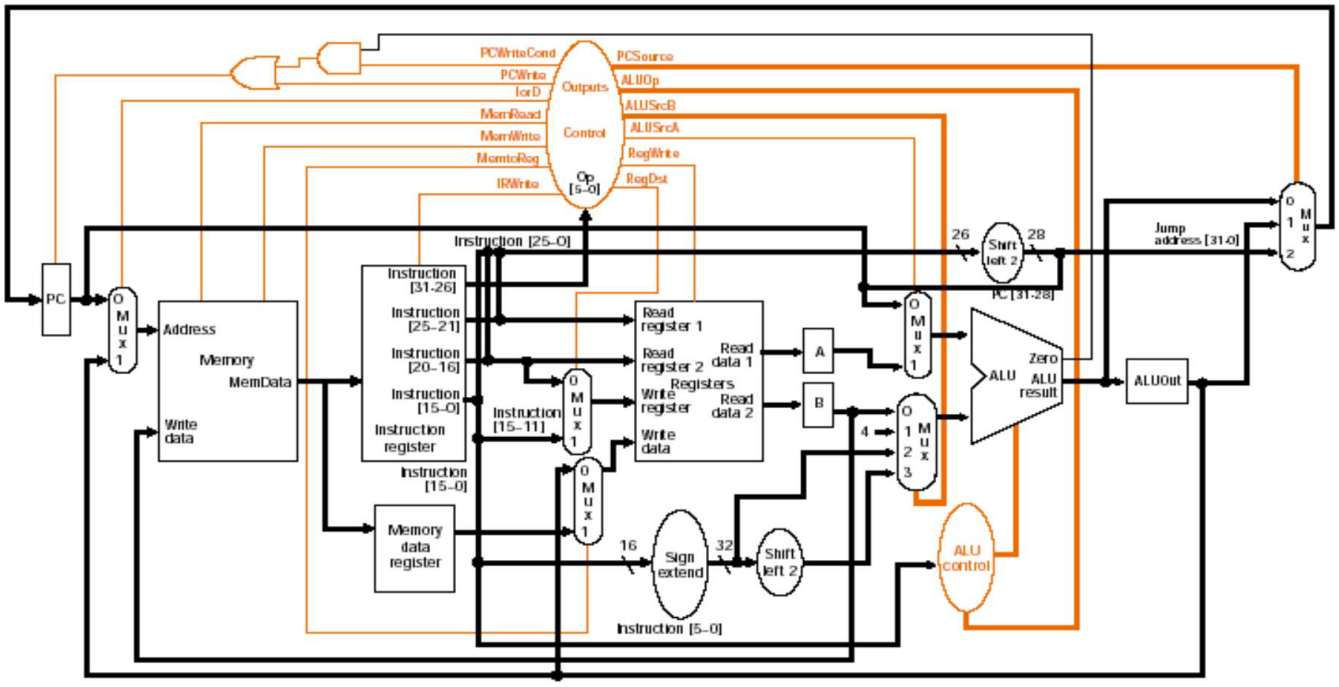
Possible sources for PC value:

Fig. 5.33

(PC + 4) directly from ALU

ALUout: result of branch calculation

Result of concatenation of left-shifted 26 bits with upper 4 bits of PC (jump)



Note that the PC is updated both unconditionally and conditionally,
so 2 control signals are needed

PCWriteCond: ANDed with ALU Zero to control PC update for branch
This result is ORed with PCWrite

PCSource: controls MUX to select input to PC
0: ALU result
1: ALUOut
2: Jump address
Why do we need both 0 and 1 inputs?

Control signals are listed in Fig. 5.34

Multi-cycle datapath: instruction execution

Breaking instruction execution into multiple clock cycles:

- Balance amount of work done in each cycle (minimizes the cycle time)
- Each step contains at most one:

Register access

Memory access

ALU operation

Any data values which are needed in a later clock cycle are stored in a register

- Major state elements: PC, register file, memory
- Temporary registers written on every cycle: A data, B data, MDR, ALUOut
- Temporary register with write control: IR

Note that we can read the current value of a destination register:

- New value doesn't get written until next clock cycle

Multiple operations can occur in parallel during same clock cycle

- Read instruction and increment PC
- Other operations occur in series during separate clock cycles
- Reading or writing standalone registers (PC, A data, B data, etc.) done in 1 cycle
- Register file access requires additional cycle: more overhead for access and control

Instruction execution steps

1. Fetch instruction from memory and compute address of next sequential instruction
2. Instruction decode and register fetch
3. R-type execution, memory address computation, or branch
4. Memory access or R-type instruction completion
5. Memory read completion

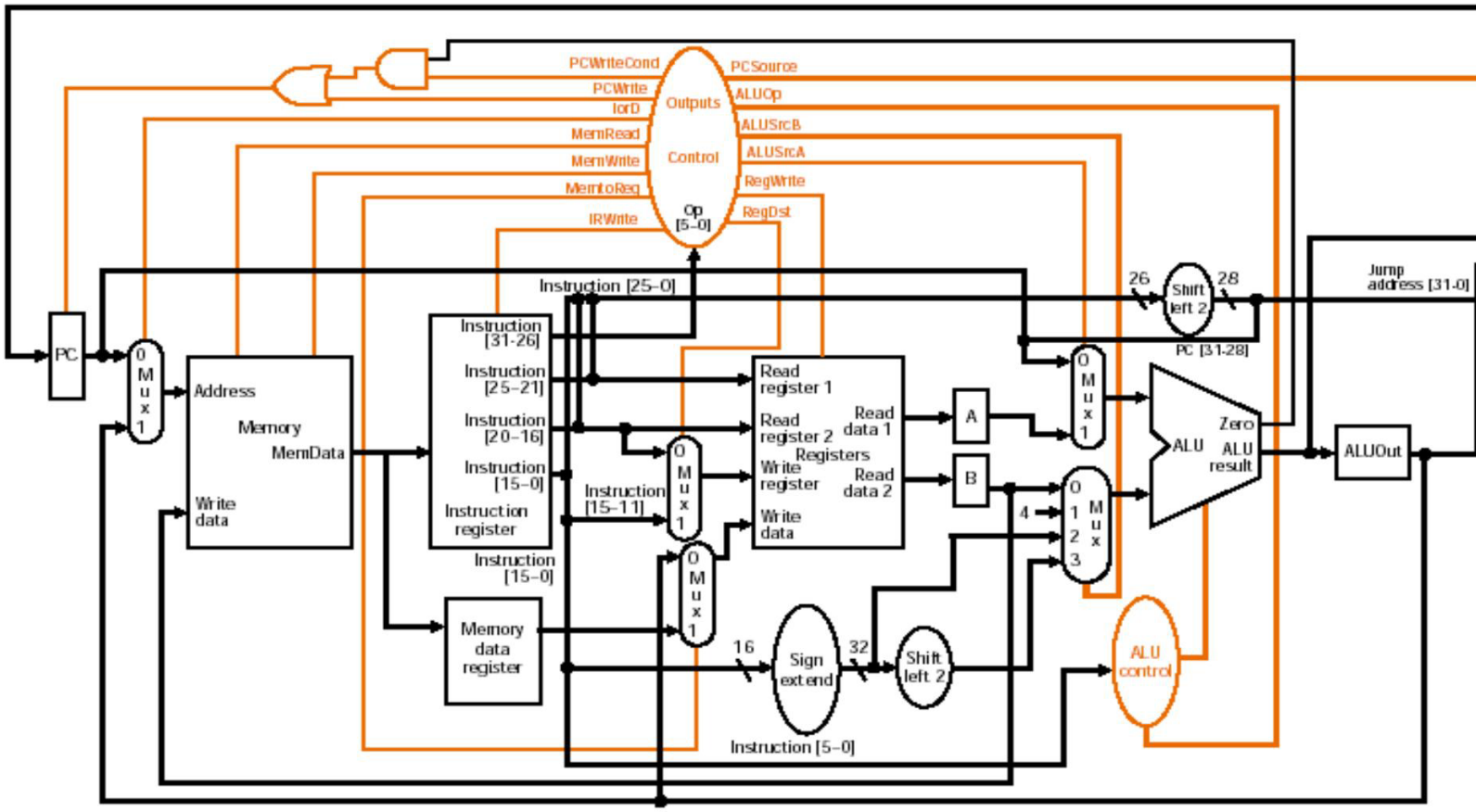
Multi-cycle datapath: instruction fetch

1. Fetch instruction from memory and compute address of next instruction

Fig. 5.33

Operation:

IR = Memory[PC];



PC = PC + 4;

Control signals needed

MemRead, IRWrite asserted

IorD set to 0 to select PC as address source

Increment PC by 4:

ALUSrcA = 0: PC to ALU

ALUSrcB = 01: 4 to ALU

ALUOp = 00: add

Store PC back

PCSource = 00: ALU result

PCWrite = 1

The memory access and PC increment can occur in parallel. Why?

Because the PC value doesn't change until the next clock cycle!

Where else is the incremented PC value stored?

ALUOut

Does this have any other effect? No

Multi-cycle datapath: decode

2. Instruction decode and register fetch

What do we know about the type of instruction so far? Nothing!

So, we can only perform operations which apply to all instructions,

or do not conflict with the actual instruction

What can we do at this point?

Read the registers from the register file into A and B

Compute branch address using ALU and save in ALUOut

But, what if the instruction doesn't use 2 registers, or it isn't a branch?

No problem; we can simply use what we need once we know what

kind of instruction we have

This is why having a regular instruction pattern is a good idea

Is this inefficient?

It does use up a little more power and generate some heat, but it doesn't cost any TIME

In fact, it means that the entire instruction can be executed in fewer clock cycles

Operation:

A = Reg[IR[25-21]];

B = Reg[IR[20-16]];

ALUOut = PC + sign_extend (IR[15-0]) << 2;

What are the control signals to determine whether to write registers A and B?

There aren't any! We can read the register file and store A and B on EVERY clock cycle.

Branch address computation:

ALUSrcA = 0: PC to ALU
ALUSrcB = 11: sign-extended/ shifted immediate to ALU
ALUOp = 00: add
These operations occur in parallel.

Multi-cycle datapath: ALU, memory address, or branch

3. R-type execution, memory address computation, or branch

ALU operates on the operands, depending on class of instruction

Memory reference:

ALUOut = A + sign_extend (IR[15-0]);

Operation: ALU creates memory address by adding operands

Control signals

ALUSrcA = 1: register A

ALUSrcB = 10: sign-extension unit output

ALUOp = 00: add

Arithmetic-logical operation (R-type):

ALUOut = A op B;

Operation:

ALU performs operation specified by function code on values in registers A, B

(Where did these operands come from?

They were read from the register file on the previous cycle.)

Control signals

ALUSrcA = 1: register A

ALUSrcB = 00: register B

ALUOp = 10: use function code bits to determine ALU control

Branch:

If (A == B) PC = ALUOut;

Operation:
ALU compares A and B. If equal, Zero output signal is set to cause branch,
and PC is updated with branch address
Control signals
ALUSrcA = 1: register A
ALUSrcB = 00: register B
ALUOp = 01: subtract
PCWriteCond = 1: update PC if Zero signal is 1
PCSource = 01: ALUOut
(What is in ALUOut, and how did it get there?
It's the branch address calculated from the previous cycle, NOT the result of A - B.
Why not? Because ALUOut is updated at the END of each cycle.)
Note that PC is actually updated twice if the branch is taken:
Output of the ALU in the previous cycle (instruction decode/ register fetch),
From ALUOut if A and B are equal
Could this cause any problems? No, because only the last value of PC
is used for the next instruction execution.
Jump:
PC = PC[31-28] || (IR[25-0] << 2);
Operation:
PC is replaced by jump address.
(Upper 4 bits of PC are concatenated with 26-bit address field of instruction
shifted left by 2 bits)
Control signals

PCSource = 10: jump address

PCWrite = 1: update PC

(Where did the jump address come from?

Output of shifter concatenated with upper 4 bits of PC.)

Multi-cycle datapath: memory access/ ALU completion

4. Memory access or R-type instruction completion

Load or store: accesses memory

Arithmetic-logical operation writes result to register

Memory reference

MDR = Memory[ALUOut]; or

Memory[ALUOut] = B;

Operation:

If operation is load, word from memory is put into MDR.

If operation is store, memory location is written with value from register B.

(Where does memory address come from?

It was computed by ALU in previous cycle.

Where does register B value come from?

It was read from register file in step 3 and also in step 2.)

Control signals

MemRead = 1 (load) or

MemWrite = 1 (store)

IorD = 1: address from ALU, not PC

What about MDR?

It's written on every clock cycle.

Arithmetic-logical operation

Reg[IR[15-11]] = ALUOut;

Operation:
ALUOut contents are stored in result register.

Control signals

RegDst = 1: use $rd field from IR for result register

RegWrite = 1: write the result register

MemtoReg = 0: write from ALUOut, not memory data

Multi-cycle datapath: memory read completion

5. Memory read completion

Value read from memory is written back to register

Reg[IR[20-16]] = MDR;

Operation:

Write the load data from MDR to target register $rt

Control signals

MemtoReg = 1: write from MDR

RegWrite = 1: write the result register

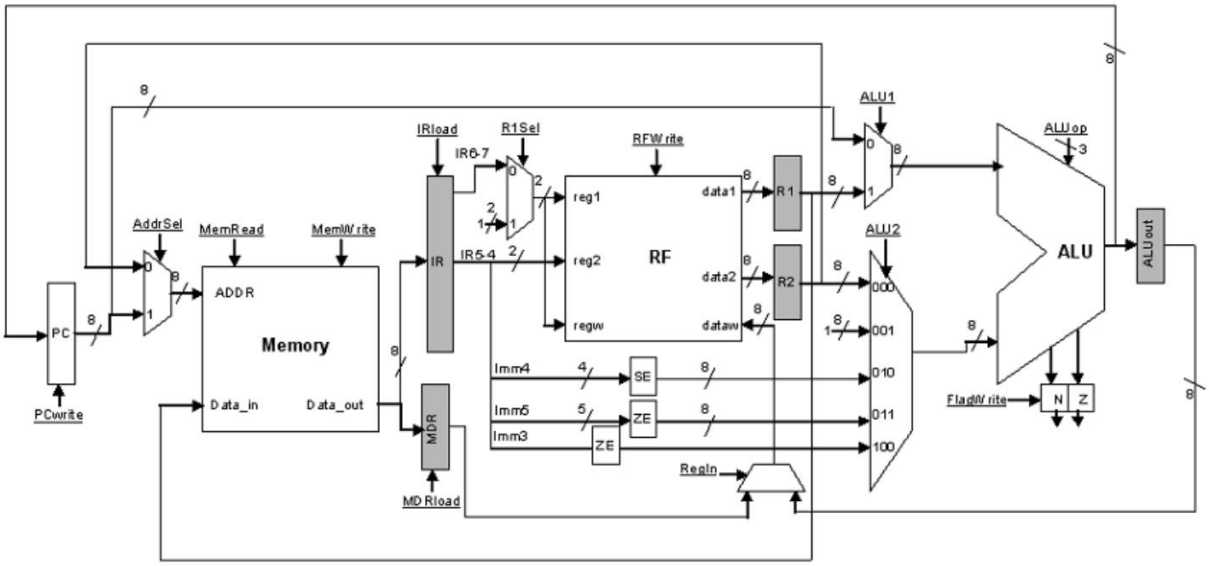
RegDst = 0: use $rt field from IR for result register

Multi-cycle datapath: summary

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches
Instruction fetch	IR = Memory[PC] PC = PC + 4		
Instruction decode/register fetch	A = Reg [IR[25–21]] B = Reg [IR[20–16]] ALUOut = PC + (sign-extend (IR[15–0]) << 2)		
Execution, address computation, branch/ jump completion	ALUOut = A op B	ALUOut = A + sign-extend (IR[15–0])	if (A == B) then PC = ALUOut
Memory access or R-type completion	Reg [IR[15–11]] = ALUOut	Load: MDR = Memory[ALUOut] or Store: Memory [ALUOut] = B	
Memory read completion		Load: Reg[IR[20–16]] = MDR	

Multicycle CPU Micro-Architecture

As with the single-cycle implementation our processor will consist of two cooperating units the datapath and the control. The key difference here is that the execution of a single instruction will take multiple cycles to complete. Accordingly, the datapath will have to change a bit. The key change will be the introduction of temporary registers to hold the outcomes that are produced at each cycle. This is best understood by looking at the schematic for the datapath. For the time being please ignore the details and focus on the grey boxes. These are the new registers:



The following temporary registers are introduced:

1. IR or Instruction Register: This is used to hold the instruction encoding after it is read from memory. A register is needed because we will use a single memory device both for data and instructions. Accordingly, its output may change during the execution of an instruction (a load will read from memory).

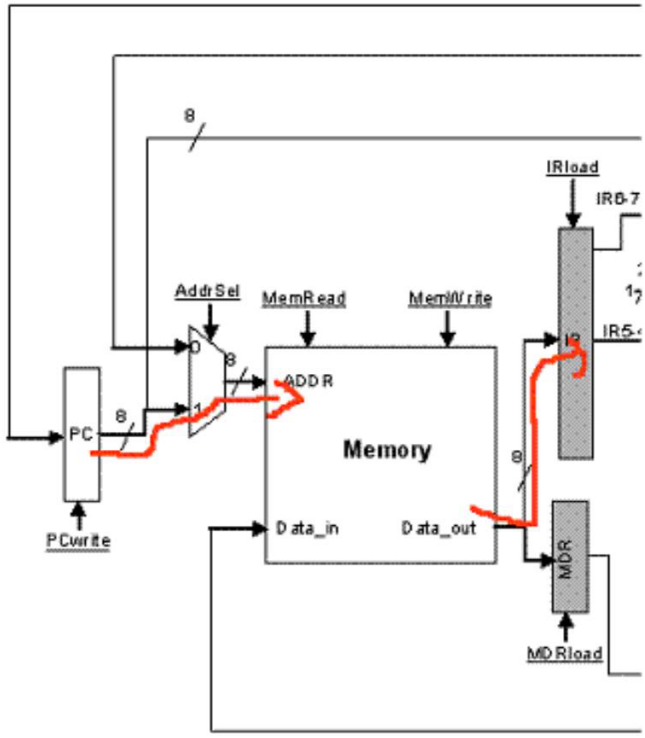
2. R1 and R2: These are used to temporarily hold the register values read from the register file.

3. AluOut: This is used to temporarily hold the result calculated by the ALU.

4. MDR or Memory Data Register: holds the value returned from memory so that it can later be written into the register file.

Let's see how this datapath was derived. We will explain what happens cycle by cycle. The first two cycles are the same for all instructions since we need to fetch the instruction from memory and then decode it (i.e., the control has to look at the opcode and decide what to do next).

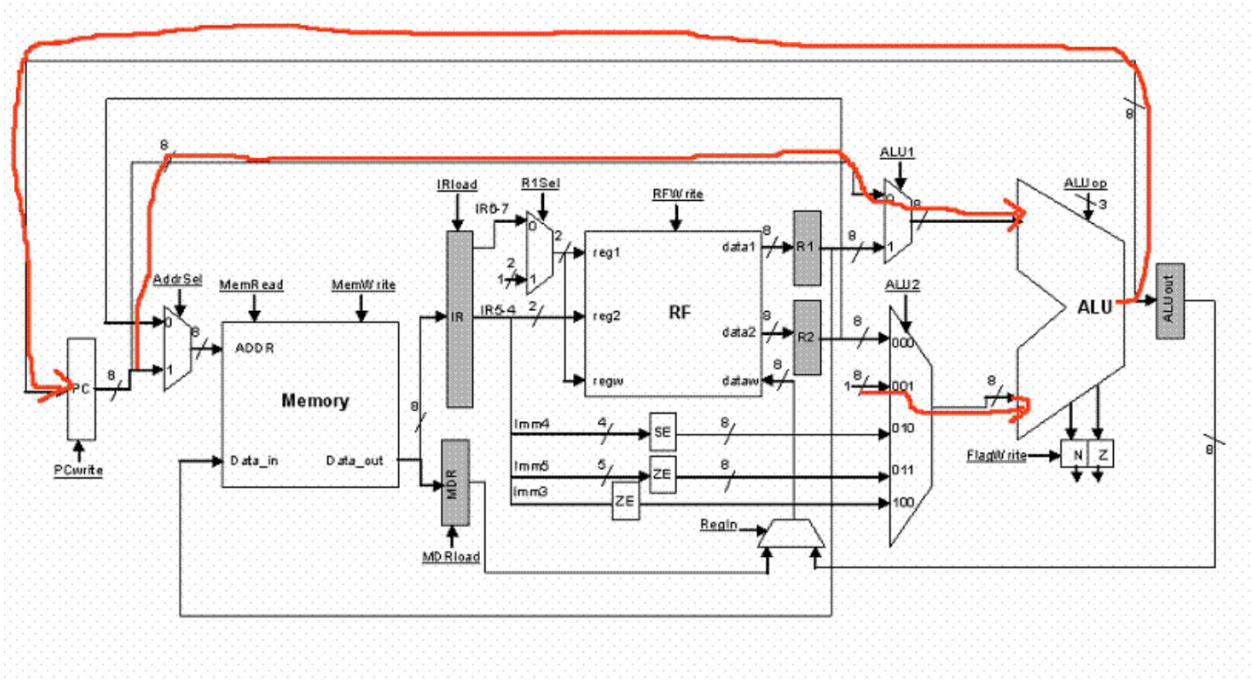
CYCLE 1: Fetching the Instruction and Incrementing the PC



The first step in executing an instruction requires fetching the instruction from memory. For this we have to send the value of the PC register to the address lines of the memory device. Assuming that the memory will respond within this first cycle, we want to store the returned value (this is the encoding of the instruction that we should execute). To do this we need to take the value from the memory's output and write into the IR register.

Because we may access the same memory device to perform a load or a store (read and write respectively) a MUX is needed at the address input so that it is possible to send either the PC or another address. So, during the first cycle we will be reading the instruction encoding from memory. This is probably a good time to also calculate PC = PC + 1 as all instructions use this (even branches require PC + 1 as part of their target calculation).

In parallel with the memory access, we send the PC value through the ALU1 mux to the ALU. As the second input to the ALU we send the number 1 (input 001 of MUX ALU2). Finally, we set ALUop to 000 (addition). As a result, the ALU will calculate PC + 1. By setting PCWrite to 1, at the end of the current clock cycle , PC will change and will become PC + 1.

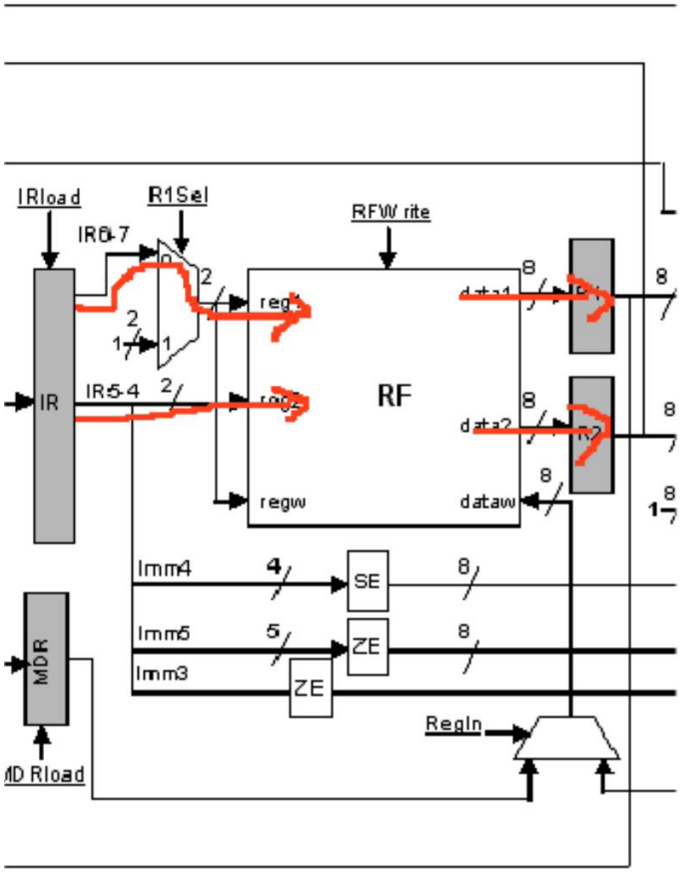


CYCLE 1 SUMMARY: In summary the following actions take place during the first cycle. This is often called the FETCH cycle.

[IR] = Mem[[PC]]

[PC] = [PC] + 1

CYCLE 2: Decoding the instruction and reading from the register file. During the second cycle, the control will be taking a look at the instruction opcode in order to decide what should happen during the next cycle. Because many instructions use the registers specified in fields R1 and R2 of the instruction we also read these registers from the register file. Note that some instructions do not use R1 or R2. In this case, we would have read registers that we do not need. While this is extra work we literally had nothing better to do during the second cycle. So, it is OK in hardware to perform actions that may be useful and later ignore the results if they are not needed. This is permissible as long as the extraneous work does not change and machine state in an irreversible way (reads do not change the register values so they are OK).



Thus at the end of the 2nd cycle, registers R1 and R2 are loaded with the values held by the registers identified by the instruction bit fields R1 and R2 respectively.

CYCLE 2 SUMMARY:

[R1] = RF[[IR7..6]]

[R2] = RF[[IR5..4]]

Instruction Decode

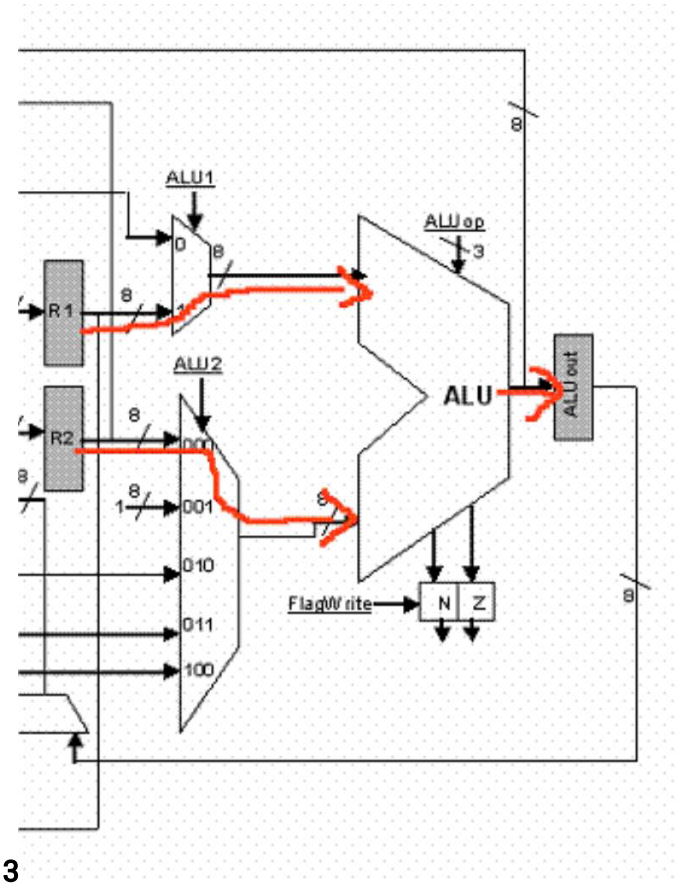
CYCLE 3 and 4

What happens after cycle 2 depends on the actual instruction. Accordingly we will consider each instruction in turn.

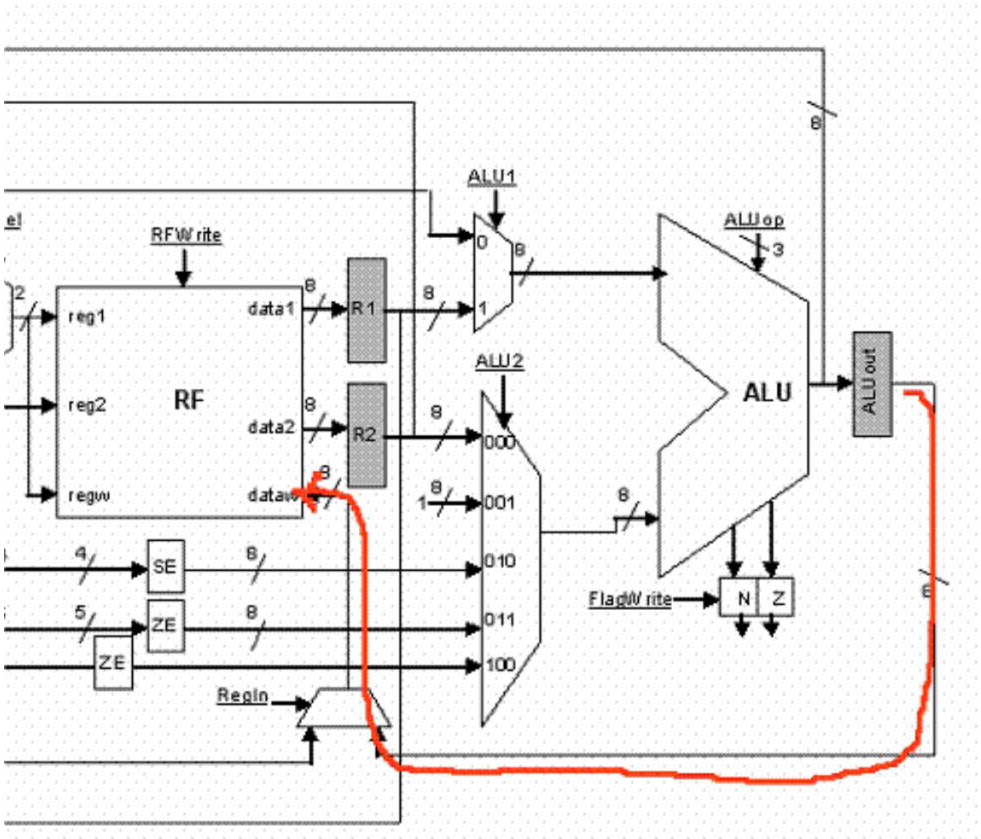
*** ADD, SUB

The execution of these three instruction proceeds into additional steps:

In cycle 3 we calculate the operation specified by the instruction and at the end store the result into ALUout. In cycle 4 we write the result into the register file:


8
ALU1
0
8
ALUop
3
R1
8
1
ALU2
ALU
ALU out
R2
8
000
8
1
001
8
010
011
100
FlagWrite
N
Z
8

Cycle 3



甚 Cycle 4

Why a Multiple Cycle CPU?

• The problem => single-cycle cpu has a cycle time long enough to complete the longest instruction in the machine

• The solution => break up execution into smaller tasks, each task taking a cycle, different instructions requiring different numbers of cycles or tasks

• Other advantages => reuse of functional units (e.g., alu, memory)