CSE 431
Computer Architecture
Fall 2015

Chapter 4A: The Single Cycle Processor

Mary Jane Irwin (www.cse.psu.edu/~mji)

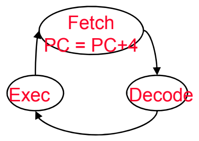
[Adapted from *Computer Organization and Design, 5th Edition*,
Patterson & Hennessy, © 2014, Morgan Kaufmann]

Reminders

- This lecture
  - MIPS datapath and control, PH 4.1-4.4
- Next lecture
  - MIPS pipelined datapaths, data hazards, PH 4.5-4.7
- Reminders
  - HW2 is due Sept 17th (our first using SimpleScalar)
  - Quiz 2 will open Sept 9th and will close midnight Sept 22nd
  - Attend one of the unix + SimpleScalar tutorial sessions in the lab (218 IST) Sept 9th and 10th from 7:30 to 9pm
  - First evening midterm exam scheduled
    - Tuesday, October 6th, 20:15 to 22:15, Location 22 Deike
    - **I should have heard from all who have conflicts by now**

The Processor: Datapath & Control

- Our implementation of the MIPS is simplified
  - memory-reference instructions: `lw, sw`
  - arithmetic-logical instructions: `add, sub, and, or, slt`
  - control flow instructions: `beq, j`
- Generic implementation
  - use the program counter (PC) to supply the instruction address and fetch the instruction from memory (and update the PC)
  - decode the instruction (and read registers)
  - execute the instruction

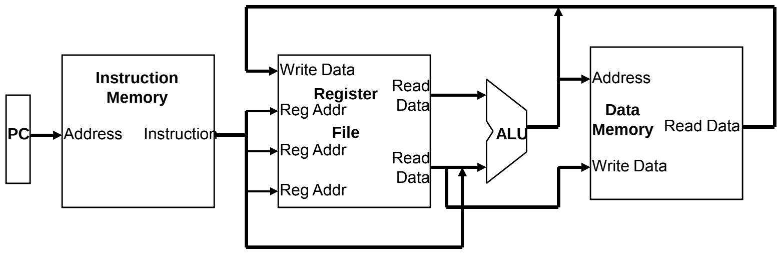


- All instructions (except `j`) use the (integer) ALU after reading the registers

How is the ALU used?
memory-reference? arithmetic? control flow?

Abstract Implementation, First Version

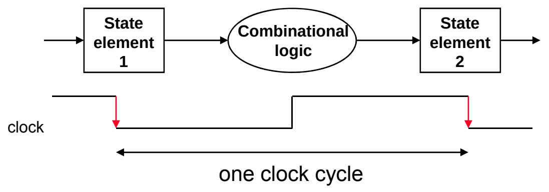
- Two types of functional units:
  - elements that operate on data values (combinational)
  - elements that contain state (sequential)



- Single cycle operation
- Split memory model – one memory for instructions (really L1I$) and one memory (really L1D$) for data

Aside: Clocking Methodologies

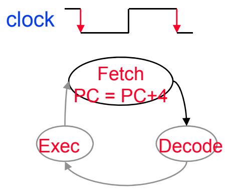
- The clocking methodology defines when data in a state element is valid and stable relative to the clock
  - State elements – a memory element such as a register
  - Edge-triggered – all state changes occur on a clock edge
- Typical execution
  - read contents of state elements -> send values through combinational logic -> write results to one or more state elements

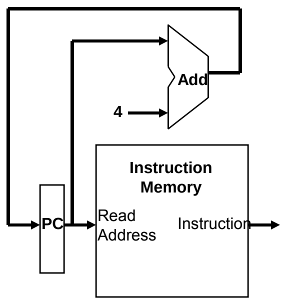


- Assumes state elements are written on every clock cycle; if not, need explicit write control signal
  - write occurs only when both the write control is asserted and the clock edge occurs

Fetching Instructions

- Fetching instructions involves
  - reading the instruction from the Instruction Memory
  - updating the PC value to be the address of the next (sequential) instruction

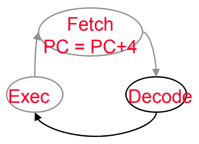




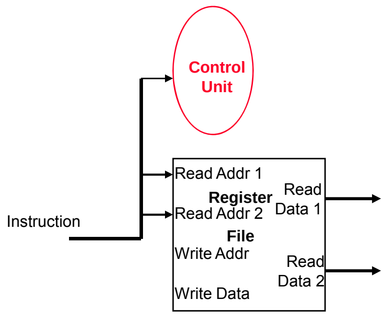
  - PC is updated every clock cycle, so it does not need an explicit write control signal just a clock signal
  - Reading from the Instruction Memory is a combinational activity, so it doesn't need an explicit read control signal

Decoding Instructions

- ❑ Decoding instructions involves
  - sending the fetched instruction's opcode and function field bits to the control unit



and



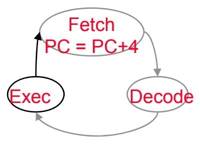
  - reading two values from the RF
    - RF addresses are contained in the instruction
    - Can read registers before the instruction is fully decoded, then ignore what you don't need. Benefits? Costs?

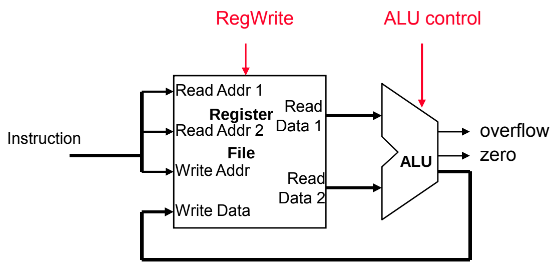
Executing R Format Operations

- ❑ R format operations (**add, sub, slt, and, or**)

	31	25	20	15	10	5	0
R-type:	**op**	**rs**	**rt**	**rd**	**shamt**	**funct**	

  - perform operation (op and funct) on values in rs and rt
  - store the result back into the RF (into location rd)

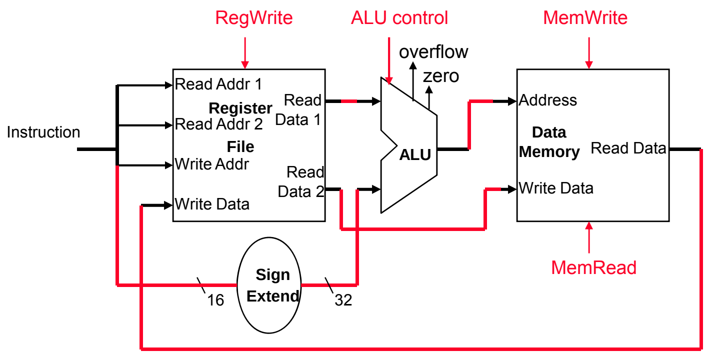




  - Note that RF is not written every cycle (e.g., **sw**), so we need an explicit write control signal (RegWrite) for RF

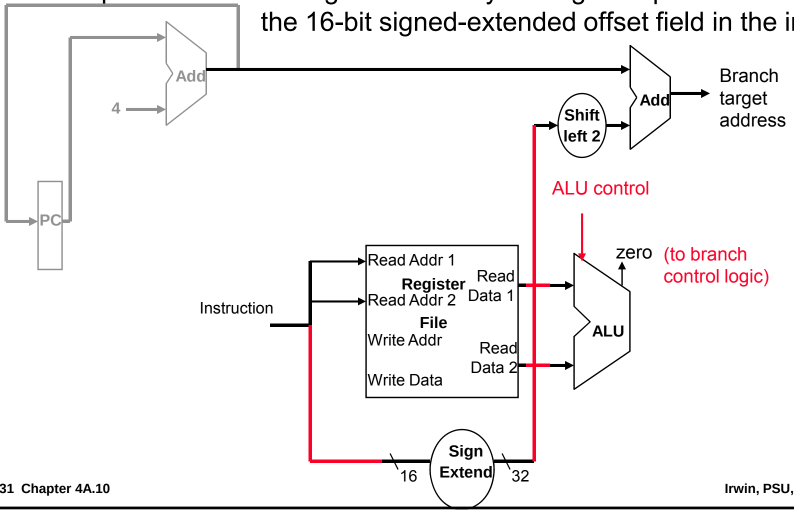
Executing Load and Store Operations

- Load and store operations involves
  - compute memory address by adding the base register (read from the RF during decode) to the 16-bit signed-extended offset field in the instruction
  - store value (read from the RF during decode) written to the Data Memory
  - load value, read from the Data Memory, written to the RF



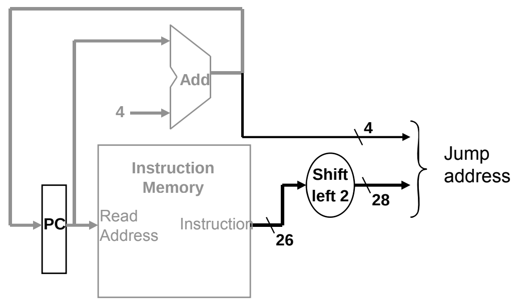
Executing Branch Operations

- Branch operations involves
  - compare the operands read from the RF during decode for equality (**zero** ALU output)
  - compute the branch target address by adding the updated PC to the 16-bit signed-extended offset field in the instr



Executing Jump Operations

- Jump operation involves
  - replace the lower 28 bits of the PC with the lower 26 bits of the fetched instruction shifted left by 2 bits

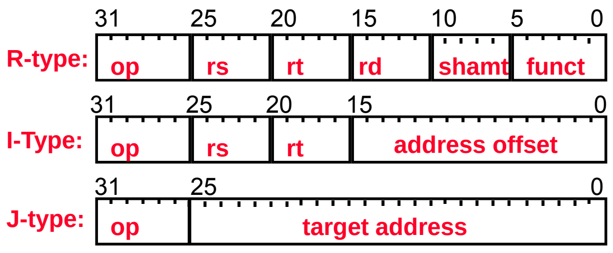


Creating a Single Datapath from the Parts

- Assemble the datapath segments and add control lines and multiplexors as needed
- Single cycle design – fetch, decode and execute each instructions in one clock cycle
  - no datapath resource can be used more than once per instruction, so some must be duplicated (e.g., separate Instruction Memory and Data Memory, several adders)
  - multiplexors needed at the input of shared elements with control lines to do the selection
  - write signals to control writing to the RF and Data Memory
- Cycle time is determined by length of the longest path

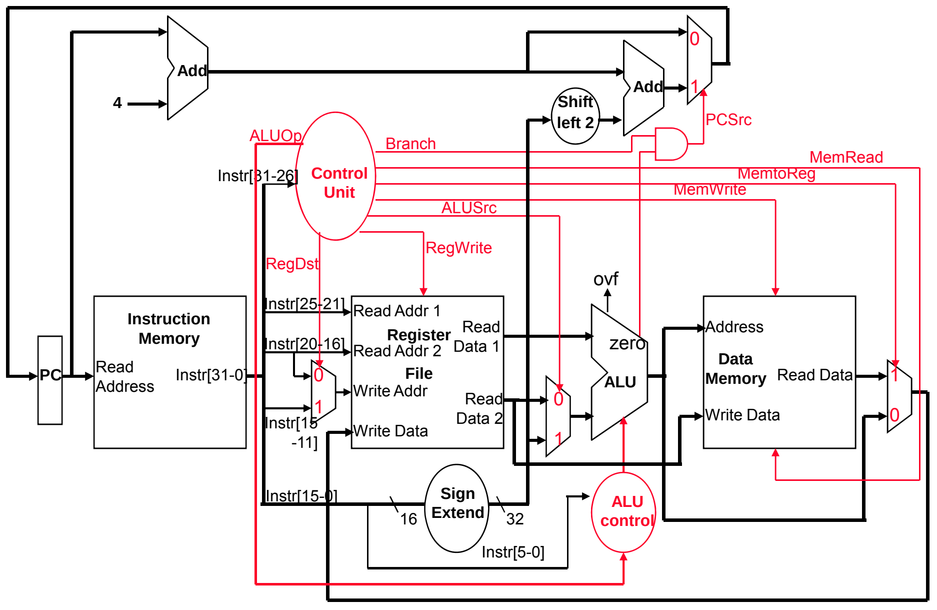
Adding the Control

- Selecting the operations to perform (ALU, RF and Memory read/write)
- Controlling the flow of data (multiplexor inputs)

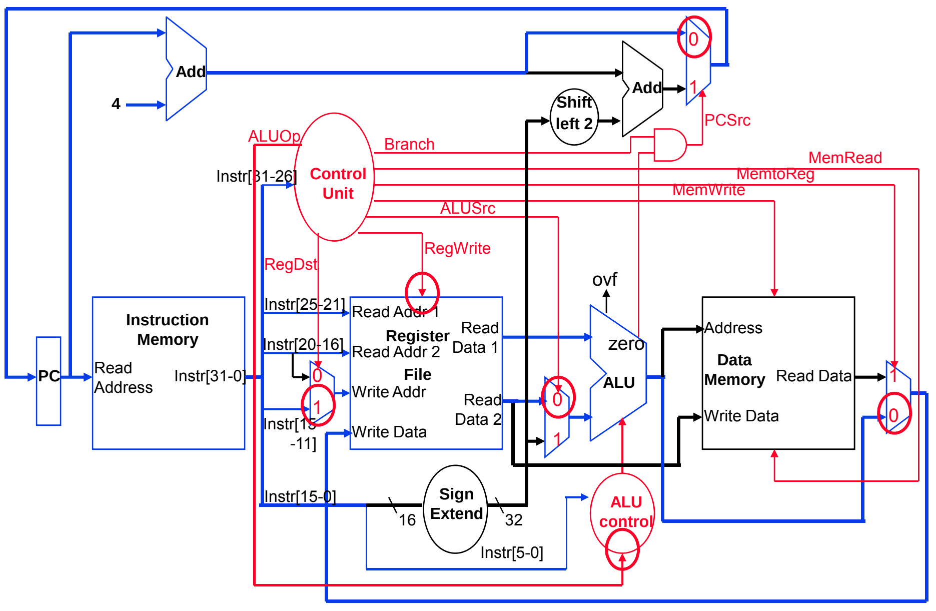


- Observations
  - op field always in bits 31-26
  - addr of registers to be read are always specified by the rs field (bits 25-21) and rt field (bits 20-16); for lw and sw rs is the base register
  - addr. of register to be written is in one of two places – in rt (bits 20-16) for lw; in rd (bits 15-11) for R-type instructions
  - offset for beq, lw, and sw always in bits 15-0

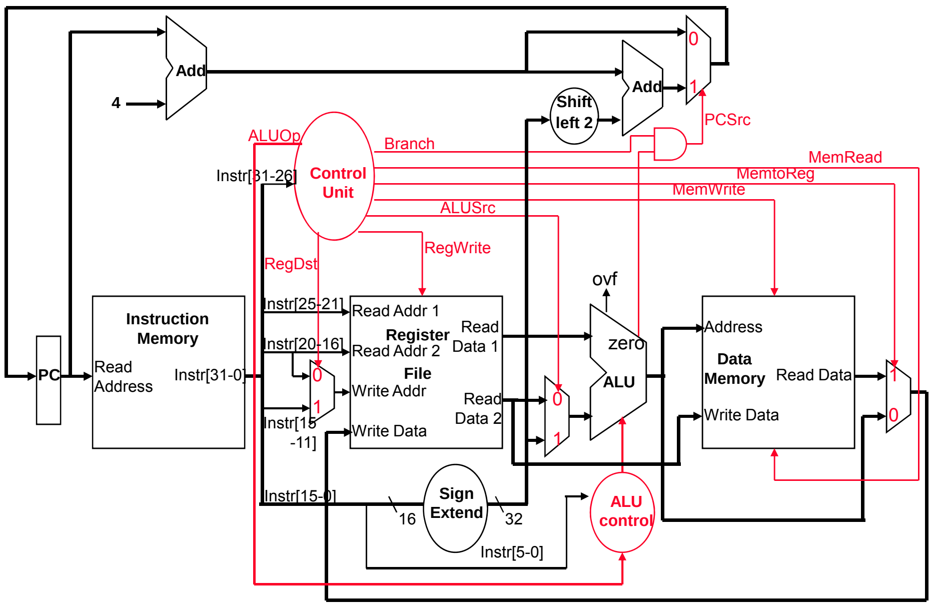
Single Cycle Datapath with Control Unit



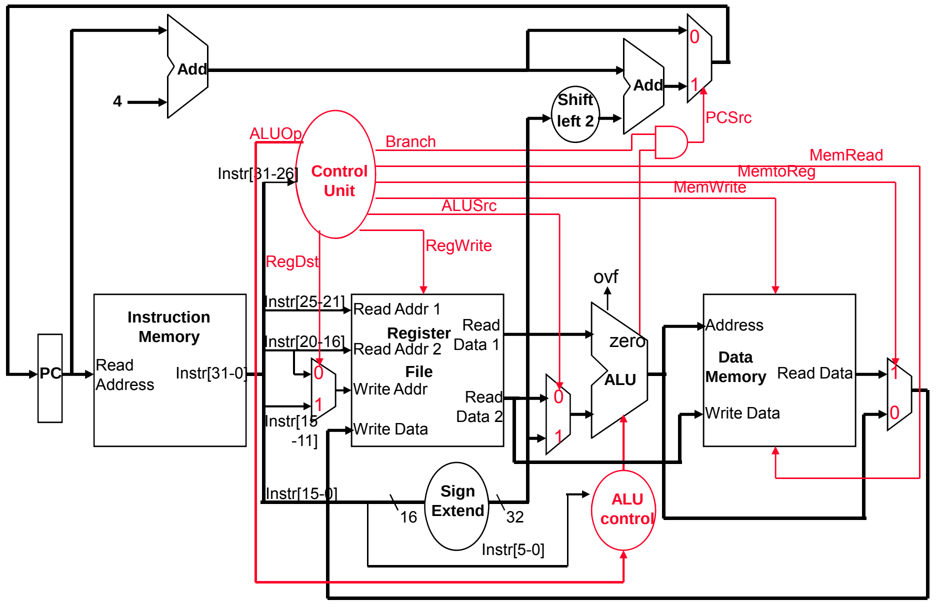
R-type Instruction Data/Control Flow



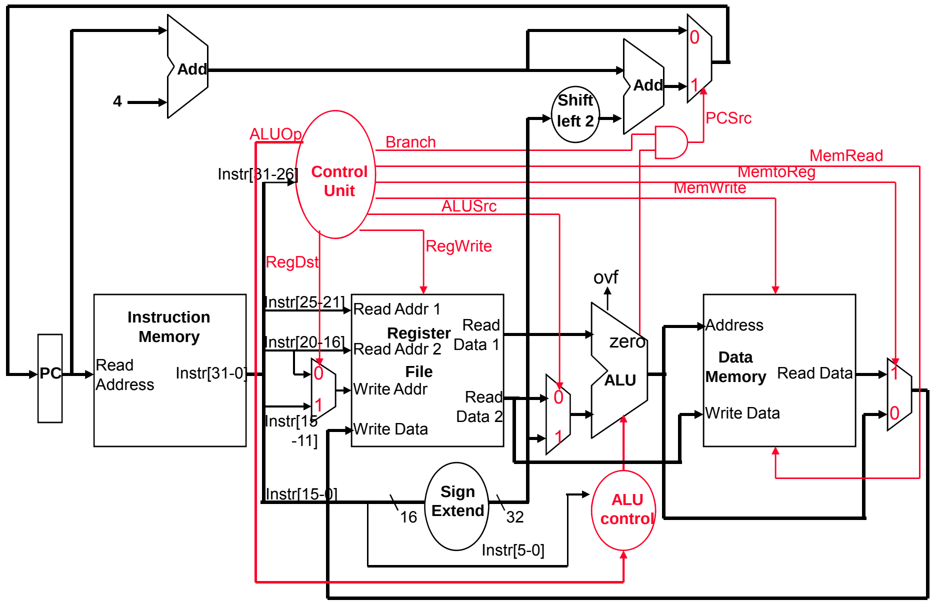
Load Word Instruction Data/Control Flow



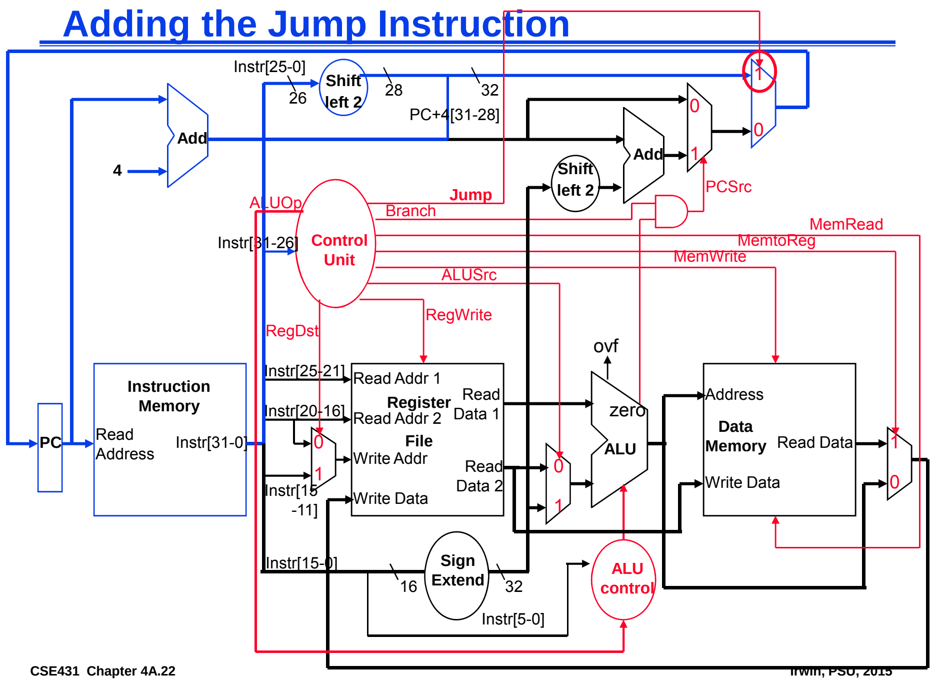
Store Word Instruction Data/Control Flow



Branch Instruction Data/Control Flow



Adding the Jump Instruction



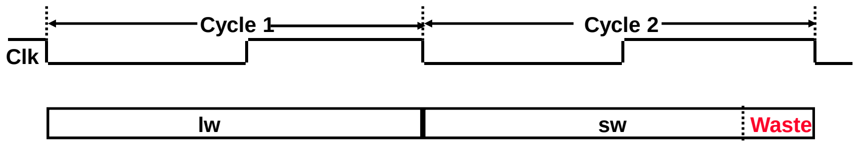
Instruction Times (Critical Paths)

- What is the clock cycle time assuming negligible delays for muxes, control unit, sign extend, PC access, shift left 2, wires, setup and hold times except:
  - Instruction and Data Memory (200 ps)
  - ALU and adders (200 ps)
  - Register File access (reads or writes) (100 ps)

Instr.	I Mem	Reg Rd	ALU Op	D Mem	Reg Wr	Total
R-type						
load						
store						
beq						
jump						

Single Cycle Disadvantages & Advantages

- Uses the clock cycle inefficiently – the clock cycle must be timed to accommodate the slowest instruction
  - especially problematic for more complex instructions like floating point multiply



- May be wasteful of area since some functional units (e.g., adders) must be duplicated since they can not be shared during a clock cycle

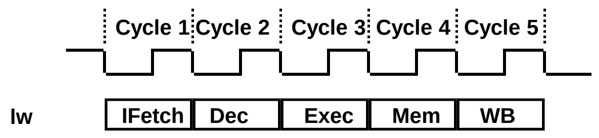
but

- Is simple and easy to understand

How Can We Make It Faster?

- Start fetching and executing the next instruction before the current one has completed
  - Pipelining – (all?) modern processors are pipelined for performance
  - Remember *the* performance equation:
    CPU time = CPI * CC * IC
- Under *ideal* conditions and with a large number of instructions, the speedup from pipelining is approximately equal to the number of pipe stages
  - A five stage pipeline is nearly five times faster because the CC is nearly five times faster
- Fetch (and execute) more than one instruction at a time
  - Superscalar processing – stay tuned

The Five Stages of Load Instruction



IFetch: Instruction Fetch and Update PC

Dec: Registers Fetch and Instruction Decode

Exec: Execute R-type; calculate memory address

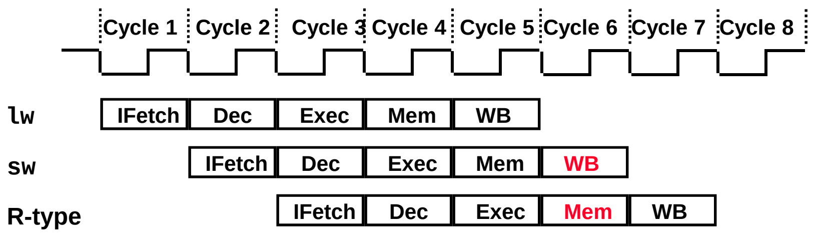
Mem: Read/write the data from/to the Data Memory

WB: Write the result data into the register file

- Single cycle – each stage is used once in each cycle
  - One active instruction per cycle (a looonnng cycle)
- Pipelined – each stage is used in each cycle
  - Multiple active instructions per cycle (a short cycle)

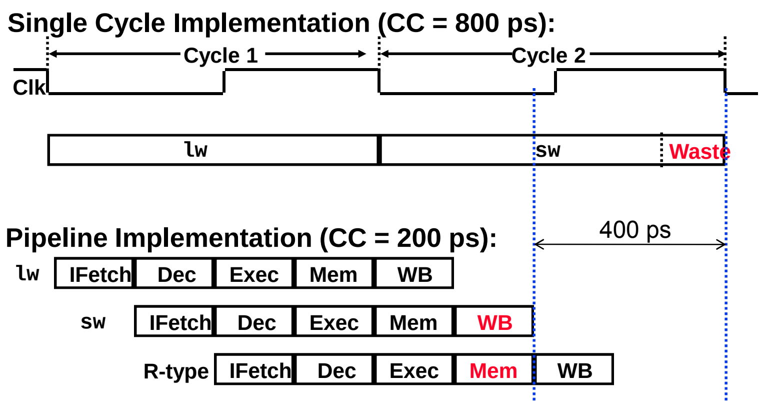
A Pipelined MIPS Processor

- Start the next instruction before the current one has completed
  - improves throughput - total amount of work done in a given time
  - instruction latency (execution time, delay time, response time - time from the start of an instruction to its completion) is *not* reduced and may increase slightly



- clock cycle (pipeline stage time) is limited by the **slowest** stage
  - for some stages don't need the whole clock cycle (e.g., WB)
- for some instructions, some stages are wasted cycles (i.e., nothing is done during that cycle for that instruction)

Single Cycle versus Pipeline



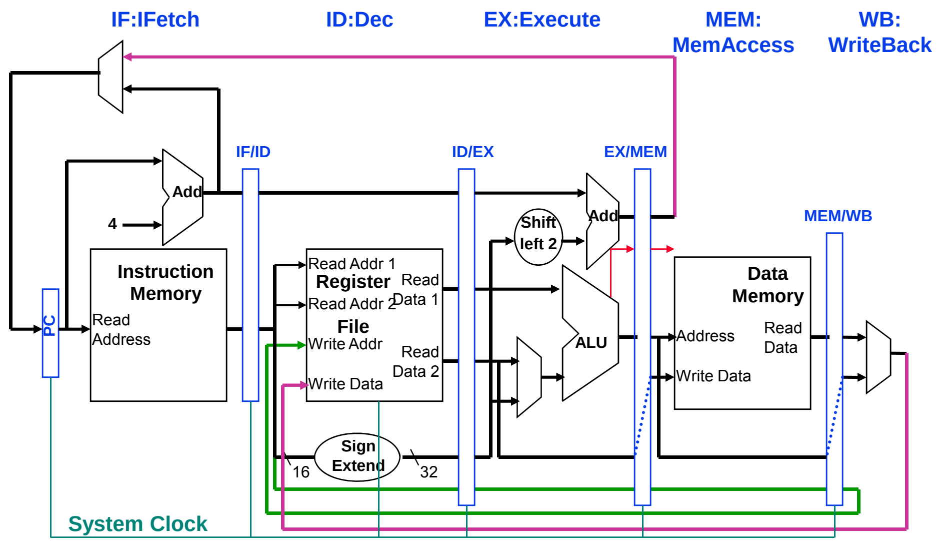
- To complete an entire instruction in the pipelined case takes 1000 ps (as compared to 800 ps for the single cycle case). Why ?
- How long does each take to complete 1,000,000 adds ?

Pipelining the MIPS ISA – What Makes It Easy?

- All instructions are the same length (32 bits)
  - can fetch in the 1st stage and decode in the 2nd stage
- Only a few instruction formats (three) with symmetry across formats
  - can begin reading register file in 2nd stage (before instruction is fully decoded) even if it turns out we don't need it
- Memory operations occur only in loads and stores
  - can use the execute stage to calculate memory addresses
- Each instruction writes at most one result (i.e., changes the machine state) and does it in the last few pipeline stages (MEM or WB)
- Operands must be aligned in memory so a single data transfer takes only one data memory access

MIPS Pipeline Datapath Additions/Mods

- State registers between each pipeline stage to isolate them



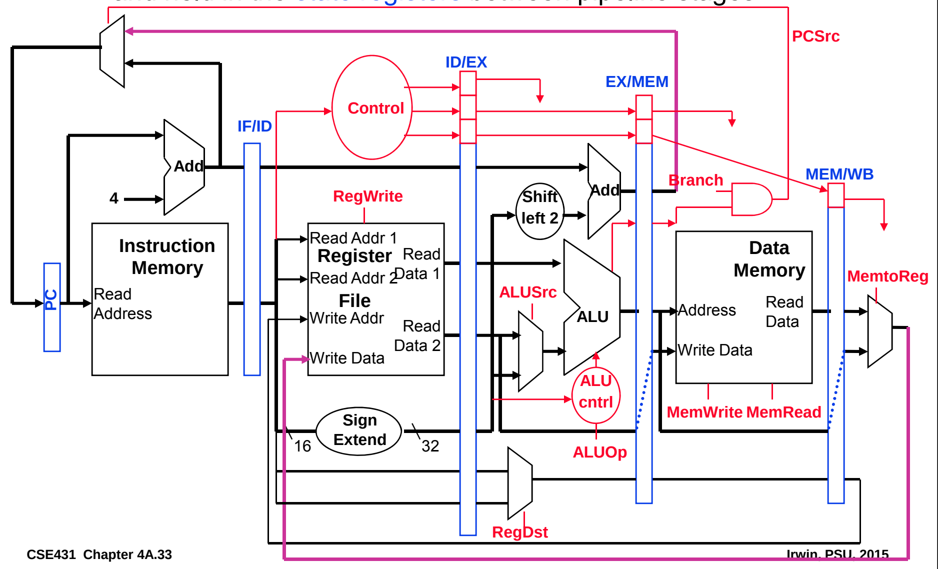
Pipeline Control

- IF Stage: read Instr Memory (always asserted) and write PC (on System Clock)
- ID Stage: decoding so no control signals ready yet

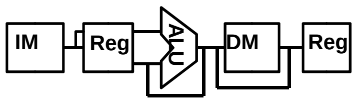
	EX Stage				MEM Stage			WB Stage	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Brch	Mem Read	Mem Write	Reg Write	Mem toReg
R	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

MIPS Pipeline Control Path Modifications

- ❑ All control signals can be determined during Decode
  - ● and held in the state registers between pipeline stages

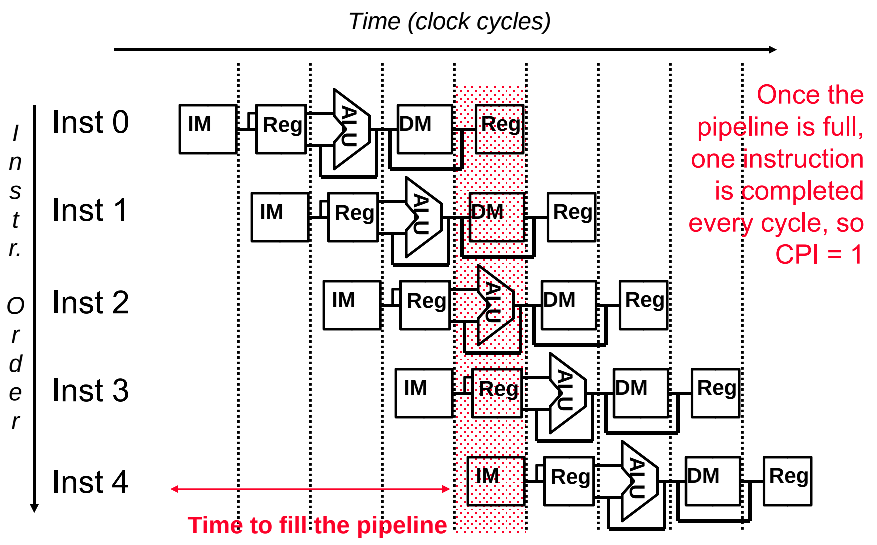


Graphically Representing MIPS Pipeline



- ❑ Can help with answering questions like:
  - ● How many cycles does it take to execute this code?
  - ● What is the ALU doing during cycle 4?
  - ● Is there a hazard, why does it occur, and how can it be fixed?

Why Pipeline? For Performance!

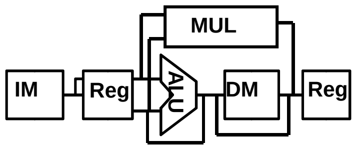

Time (clock cycles)
Instr. Order
Inst 0
Inst 1
Inst 2
Inst 3
Inst 4
IM
Reg
ALU
DM
Reg
Once the pipeline is full, one instruction is completed every cycle, so CPI = 1
Time to fill the pipeline

Pipelining the MIPS ISA – What Makes It Hard?

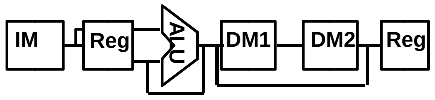
- Pipeline Hazards
  - structural hazards: attempt to use the same resource by two different instructions at the same time
  - data hazards: attempt to use data before it is ready
    - An instruction's source operand(s) are produced by a prior instruction still in the pipeline
  - control hazards: attempt to make a decision about program control flow before the condition has been evaluated and the new PC target address calculated
    - branch and jump instructions, exceptions
- Dependencies backward in time cause hazards
- Can usually resolve hazards by waiting
  - pipeline control must detect the hazard
  - and take action to resolve hazards

Other Pipeline Structures Are Possible

- ❑ What about the (slow) multiply operation?
  - Make the clock twice as slow or …
  - let it take two cycles (since it doesn't use the DM stage)



- ❑ What if the data memory access is twice as slow as the instruction memory?
  - make the clock twice as slow or …
  - let data memory access take two cycles (and keep the same clock rate)



Summary

- ❑ All modern day processors use pipelining
- ❑ Pipelining doesn't help the latency of any single instruction, it helps the throughput of the entire workload
- ❑ Potential speedup: a CPI of 1 and fast a CC
- ❑ Pipeline rate limited by slowest pipeline stage
  - Unbalanced pipe stages makes for inefficiencies
  - The time to "fill" the pipeline and the time to "drain" it can impact speedup for deep pipelines and short code runs which occur when there are a lot of branch instructions
- ❑ Must detect and resolve hazards
  - Stalling negatively affects CPI (makes CPI less than the ideal of 1)

Reminders

- Next lecture
  - MIPS pipelined datapaths, data hazards, PH 4.5-4.7

- Reminders
  - HW2 is due Sept 17th (our first using SimpleScalar)
  - Quiz 2 will open Sept 9th and will close midnight Sept 22nd
  - Attend one of the unix + SimpleScalar tutorial sessions in the lab (218 IST) Sept 9th and 10th from 7:30 to 9pm
  - First evening midterm exam scheduled
    - Tuesday, October 6th, 20:15 to 22:15, Location 22 Deike
    - **I should have heard from all who have conflicts by now**