Guerrilla 4: Pipelining Datapath

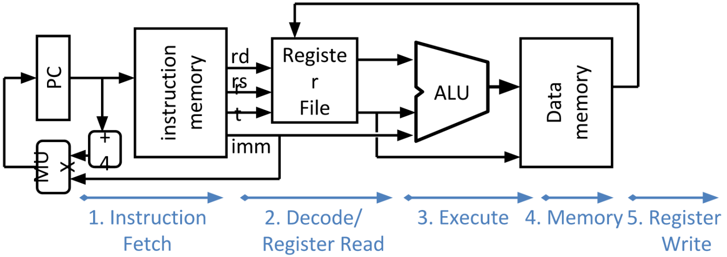
Peijie, Connor, Sandy

July 13, 2016

MIPS CPU Datapath

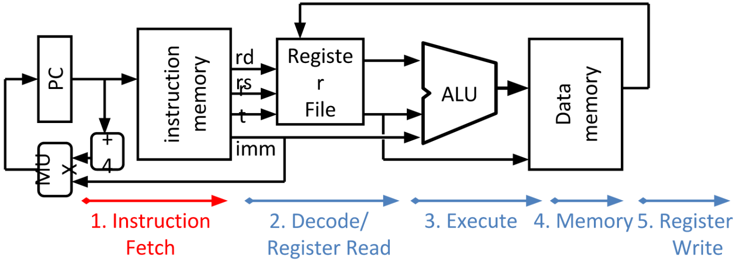
Pipeling

MIPS CPU Datapath



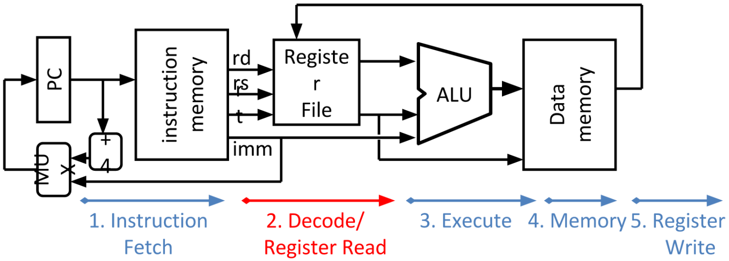
- Break up the process of “executing an instruction”
  - Smaller phases easier to design and modify independently

MIPS CPU Datapath



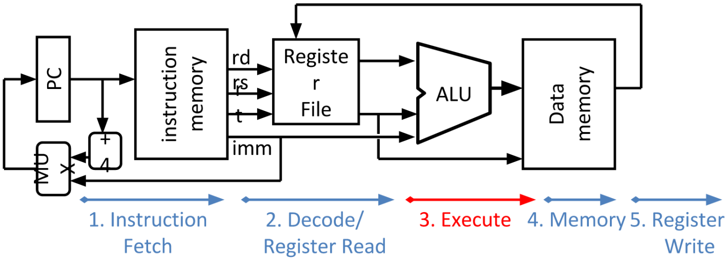
- Phase 1: *Instruction Fetch* (IF)
  - Fetch 32-bit instruction from memory
  - Increment PC (PC = PC + 4)

MIPS CPU Datapath



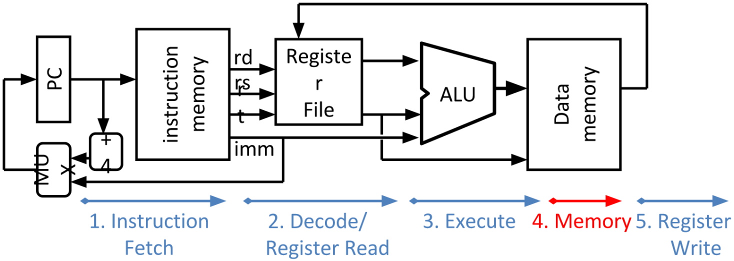
- Phase 2: *Instruction Decode* (ID)
  - Read the opcode and appropriate fields from the instruction
  - Gather all necessary registers values from Register File

MIPS CPU Datapath



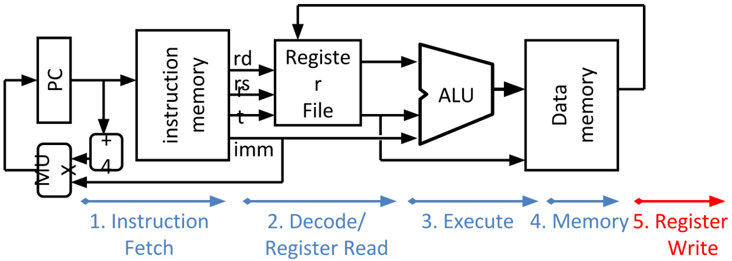
- Phase 3: *Execute* (EX)
  - ALU performs operations: arithmetic (+,-,*,/), shifting, logical (&,|), comparisons (slt,==)
  - Also calculates addresses for loads and stores

MIPS CPU Datapath



- Phase 4: *Memory Access* (MEM)
  - Only load and store instructions do anything during this phase; the others remain idle or skip this phase
  - Should hopefully be fast due to caches (later in course)

MIPS CPU Datapath



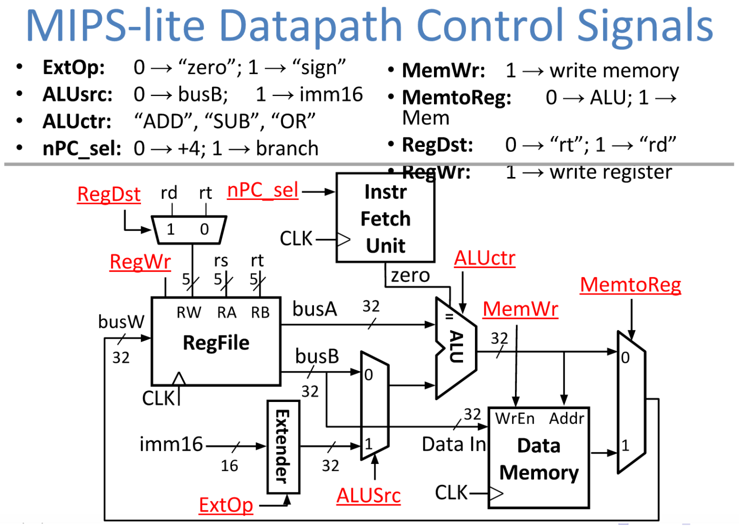
- Phase 5: *Register Write* (WB for "write back")
  - Write the instruction result back into the Register File
  - Those that don't (e.g. `sw`, `j`, `beq`) remain idle or skip this phase

Datapath Walkthrough for LW

lw $t0, 16($s1)

- IF: fetch this instruction, increment PC by 4
- ID: decode as lw, read $s1
- EX: add 16 to the value retrieved in ID to compute address
- MEM: read memory address computed in EX
- WB: write value fetched in MEM into $t0

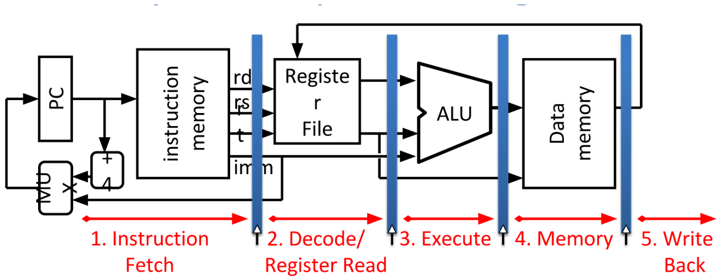
Control Signals



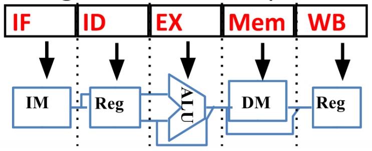
MIPS-lite Datapath Control Signals

- **ExtOp:** 0 → "zero"; 1 → "sign"
- **ALUsrc:** 0 → busB; 1 → imm16
- **ALUctr:** "ADD", "SUB", "OR"
- **nPC_sel:** 0 → +4; 1 → branch
- **MemWr:** 1 → write memory
- **MemtoReg:** 0 → ALU; 1 → Mem
- **RegDst:** 0 → "rt"; 1 → "rd"
- **RegWr:** 1 → write register

Pipelined Datapath



- Use datapath figure below to represent pipeline:



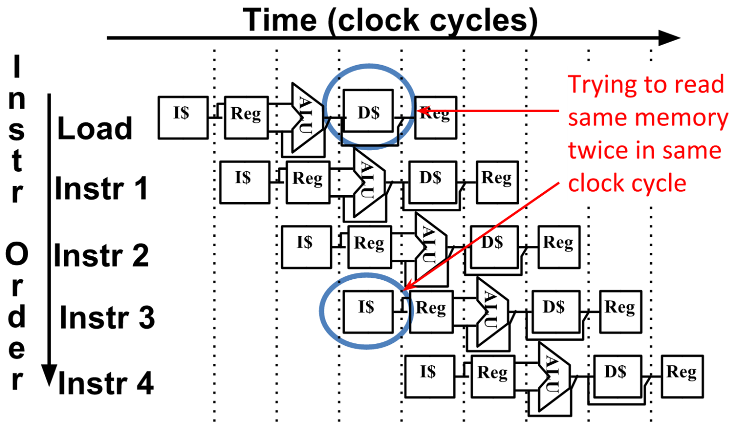
MIPS CPU Datapath

- Latency = time to complete one task
- Bandwidth/Throughput = tasks completed per unit time
- Pipelining doesn't help latency of single task, just throughput of entire workload
- Potential speedup = number of pipelined stages

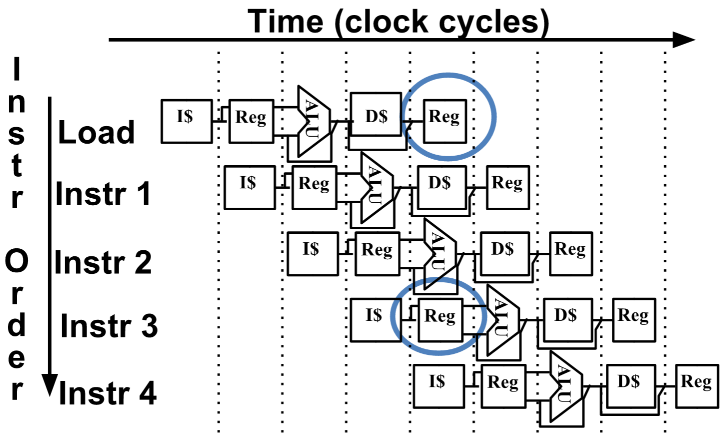
Pipelining Hazard

- 1. Structural Hazard
- 2. Data Hazard
- 3. Control Hazard

Structual Hazard

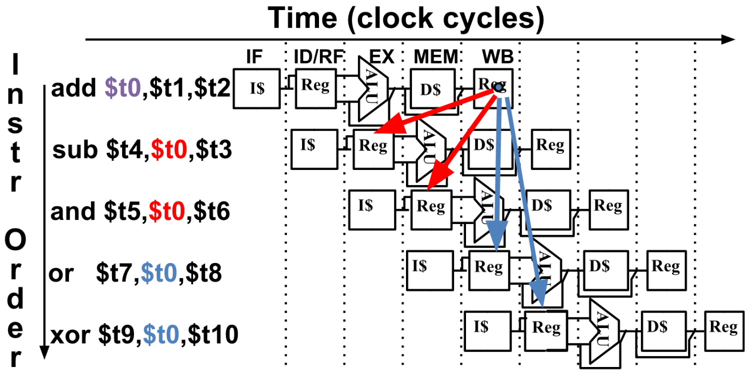


Structual Hazard



Data Hazard - Forwarding

- Data-flow *backward* in time are hazards



Data Hazard - Forwarding

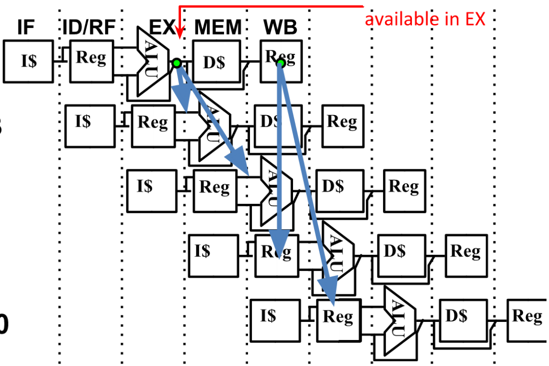
add $t0,$t1,$t2

sub $t4,$t0,$t3

and $t5,$t0,$t6

or $t7,$t0,$t8

xor $t9,$t0,$t10



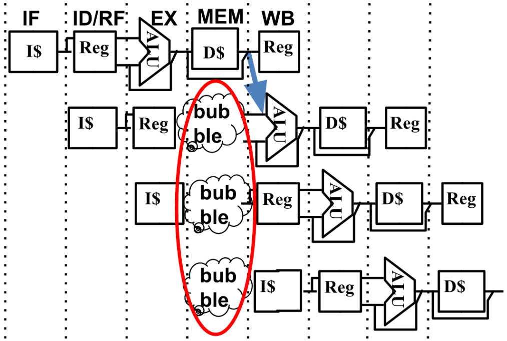
Data Hazard - Load Delay Slot

lw $t0, 0($t1)

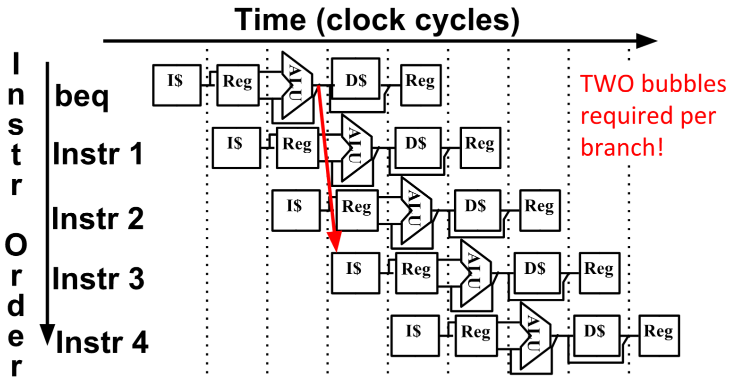
sub $t3,$t0,$t2

and $t5,$t0,$t4

or $t7,$t0,$t6



Control Hazard



Control Hazard - Branch

- 1. insert branch comparator in ID stage — only one nop is needed
- 2. Branch Prediction — Flush pipeline if prediction wrong
- 3. Branch Delay Slot
- ** Jump also has a delay slot