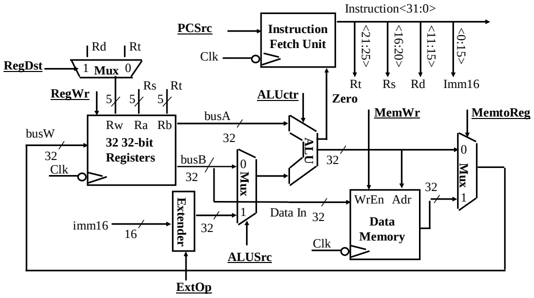
Review Day 3:
Single Cycle, Pipeline, Hazards

CS61C Summer 2004

Pooya Pakzad

Recap: A Single Cycle Datapath

- Rs, Rt, Rd and Imed16 hardwired into datapath from Fetch Unit
- We have everything except control signals (underline)



Meaning of the Control Signals

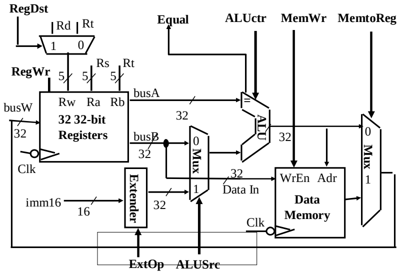
- ExtOp: “zero”, “sign”
- ALUsrc: 0 ⇒ regB; 1 ⇒ immed
- ALUctr: “add”, “sub”, “or”

° **MemWr: 1 ⇒ write memory**

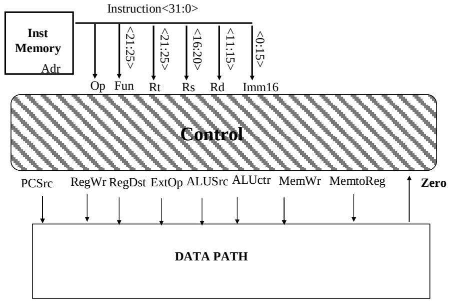
° **MemtoReg: 0 ⇒ ALU; 1 ⇒ Mem**

° **RegDst: 0 ⇒ “rt”; 1 ⇒ “rd”**

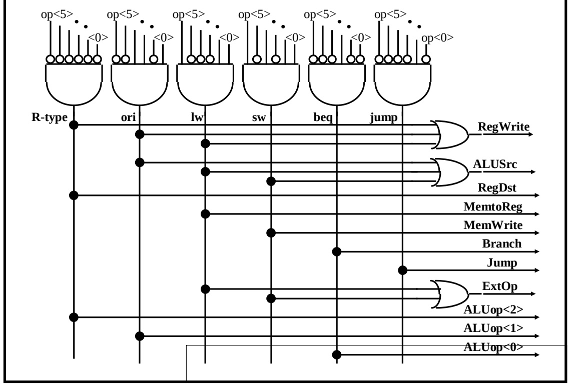
° **RegWr: 1 ⇒ write register**



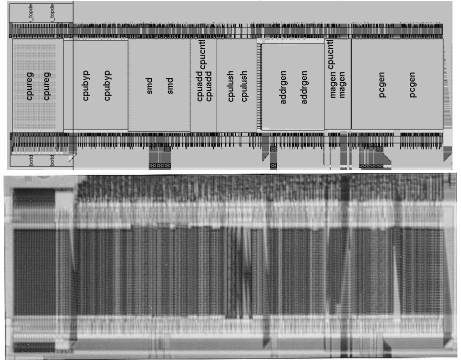
Decode -> Control->Data Path



PLA Implementation of the Main Control



A Real MIPS Datapath (CNS T0)



Drawback of this Single Cycle Processor

- Long cycle time:
  - Cycle time must be long enough for the load instruction:

    PC's Clock -to-Q +
    Instruction Memory Access Time +
    Register File Access Time +
    ALU Delay (address calculation) +
    Data Memory Access Time +
    Register File Setup Time +
    Clock Skew
- Cycle time for load is much longer than needed for all other instructions

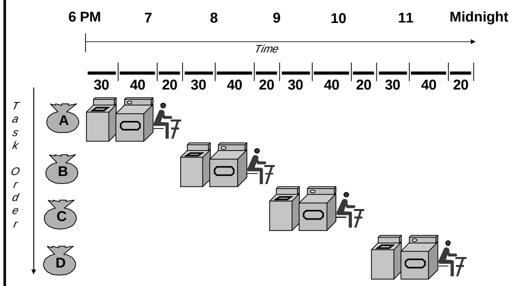
Problems

- Disadvantages of the Single Cycle Processor
  - Long cycle time
  - Cycle time is too long for all instructions except the Load
  - No reuse of hardware
- Multiple Cycle Processor:
  - Divide the instructions into smaller steps
  - Execute each step (instead of the entire instruction) in one cycle
- Partition datapath into equal size chunks to minimize cycle time

Pipelining: Big Idea

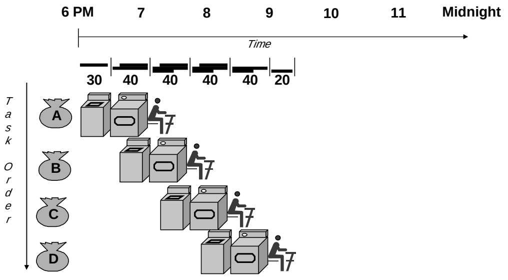
- Reduce CPI by overlapping many instructions
  - Average throughput of approximately 1 CPI with fast clock
  - Divide the instructions into smaller steps
  - Execute each step (instead of the entire instruction) in one cycle
  - Partition datapath into equal size chunks to minimize cycle time
- Utilize capabilities of the Datapath
  - start next instruction while working on the current one
  - limited by length of longest stage (plus fill/flush)
  - detect and resolve hazards
- What makes it easy
  - all instructions are the same length
  - just a few instruction formats
  - memory operands appear only in loads and stores
- What makes it hard?
  - structural hazards: suppose we had only one memory
  - control hazards: need to worry about branch instructions
  - data hazards: an instruction depends on a previous instruction

Sequential Laundry



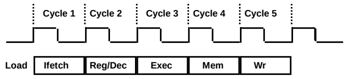
- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?

Pipelined Laundry: Start work ASAP



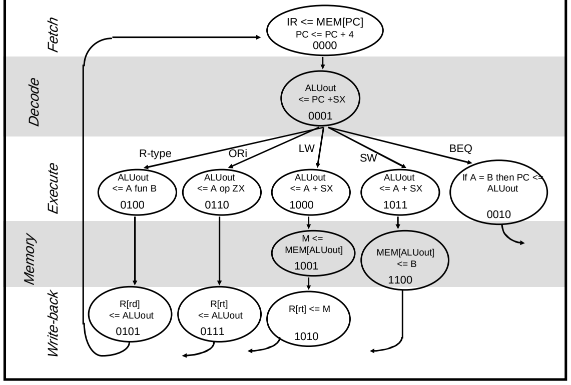
- Pipelined laundry takes 3.5 hours for 4 loads

The Five Stages of Load



- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec: Calculate the memory address
- Mem: Read the data from the Data Memory
- Wr: Write the data back to the register file

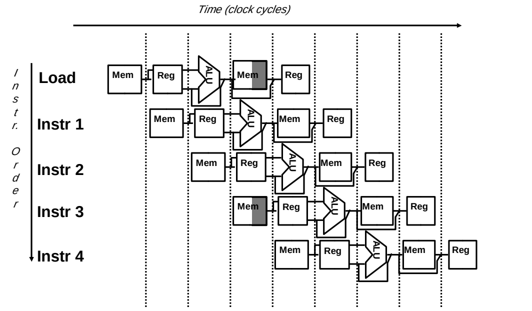
Note: These 5 stages were there all along!



Can pipelining get us into trouble?

- Yes: Pipeline Hazards
  - structural hazards: attempt to use the same resource two different ways at the same time
    - E.g., combined washer/dryer would be a structural hazard or folder busy watching TV
  - control hazards: attempt to make a decision before condition is evaluated
    - E.g., washing football uniforms and need to get proper detergent level; need to see after dryer before next load in
    - branch instructions
  - data hazards: attempt to use item before it is ready
    - E.g., one sock of pair in dryer and one in washer; can't fold until get sock from washer through dryer
    - instruction depends on result of prior instruction still in the pipeline
- Can always resolve hazards by waiting
  - pipeline control must detect the hazard
  - take action (or delay action) to resolve hazards

Single Memory is a Structural Hazard

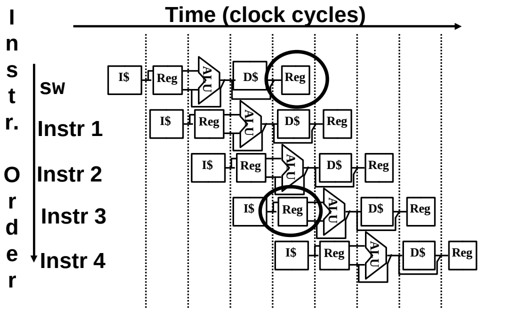


Detection is easy in this case! (right half highlight means read, left half write)

Structural Hazard: Separate Caches

- Solution:
  - infeasible and inefficient to create second memory
  - so simulate this by having <u>two Level 1 Caches</u>
  - have both an L1 <u>Instruction Cache</u> and an L1 <u>Data Cache</u>
  - L2 Cache can be unified.
  - need more complex hardware to control when both caches miss

Structural Hazard #2: Registers (1/2)



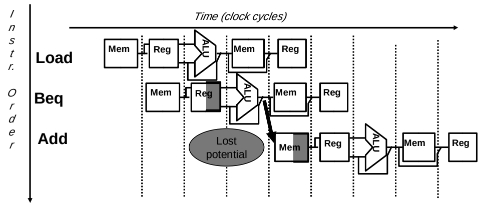
Can't read and write to registers simultaneously

Structural Hazard #2: Registers (2/2)

- Fact: Register access is *VERY* fast: takes less than half the time of ALU stage
- Solution: introduce convention
  - always Write to Registers during first half of each clock cycle
  - always Read from Registers during second half of each clock cycle
  - Result: can perform Read and Write during same clock cycle

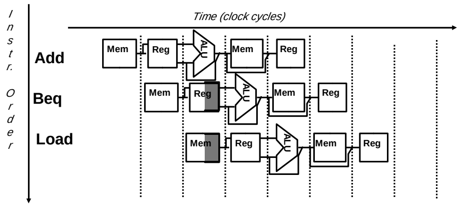
This is what we have talked about as *internal forwarding* in the regFile

Control Hazard Solution #1: Stall



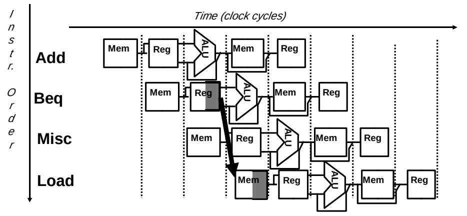
- Stall: wait until decision is clear
- Impact: 2 lost cycles (i.e. 3 clock cycles for Beq instruction above) => slow
- Move decision to end of decode
  - save 1 cycle per branch, may stretch clock cycle

Control Hazard Solution #2: Predict



- Predict: guess one direction then back up if wrong
- Impact: 0 lost cycles per branch instruction if guess right, 1 if wrong (right ~ 50% of time)
  - Need to "Squash" and restart following instruction if wrong
- More dynamic schemes: history of branch behavior (~90-99%)

Control Hazard Solution #3: Delayed Branch



- Delayed Branch: Redefine branch behavior (takes place after next instruction)
- Impact: 0 clock cycles per branch instruction if can find instruction to put in "slot" (~50% of time)
- As launch more instruction per clock cycle, less useful

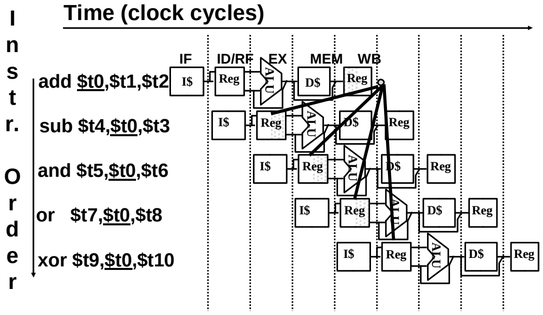
Data Hazards (1/2)

- Consider the following sequence of instructions

```
add $t0, $t1, $t2
sub $t4, $t0 ,$t3
and $t5, $t0 ,$t6
or  $t7, $t0 ,$t8
xor $t9, $t0 ,$t10
```

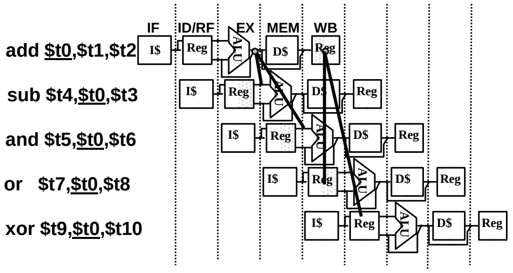
Data Hazards (2/2)

Dependencies backwards in time are hazards

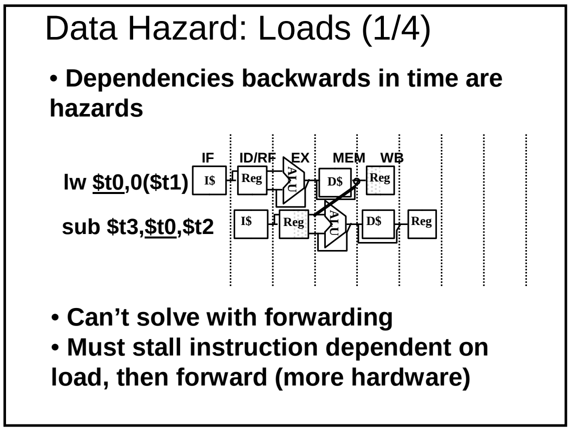


Data Hazard Solution: Forwarding

- **Forward result from one stage to another**



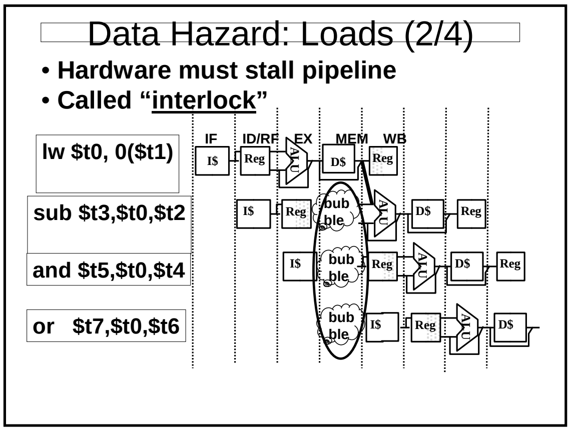
"or" hazard solved by register hardware



Data Hazard: Loads (1/4)

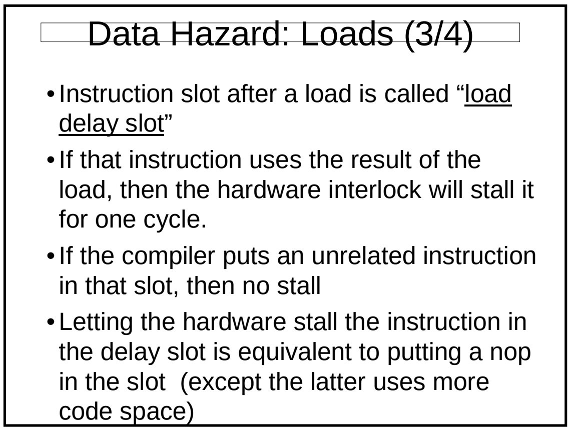
- **Dependencies backwards in time are hazards**

- **Can't solve with forwarding**
- **Must stall instruction dependent on load, then forward (more hardware)**



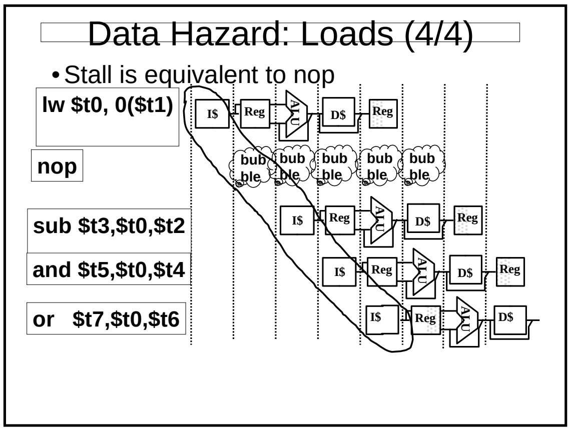
Data Hazard: Loads (2/4)

- **Hardware must stall pipeline**
- **Called "interlock"**



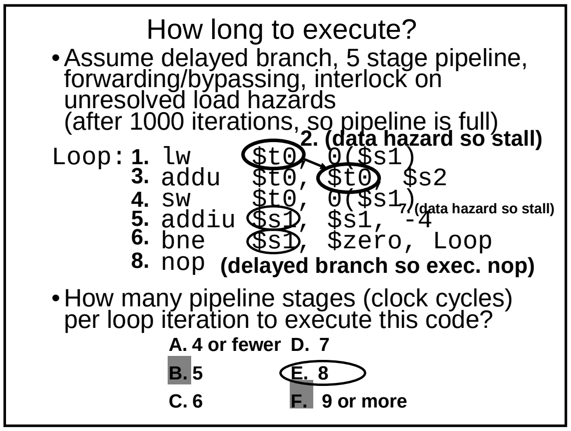
Data Hazard: Loads (3/4)

- Instruction slot after a load is called "load delay slot"
- If that instruction uses the result of the load, then the hardware interlock will stall it for one cycle.
- If the compiler puts an unrelated instruction in that slot, then no stall
- Letting the hardware stall the instruction in the delay slot is equivalent to putting a nop in the slot (except the latter uses more code space)



Data Hazard: Loads (4/4)

- Stall is equivalent to nop



How long to execute?

- Assume delayed branch, 5 stage pipeline, forwarding/bypassing, interlock on unresolved load hazards (after 1000 iterations, so pipeline is full)

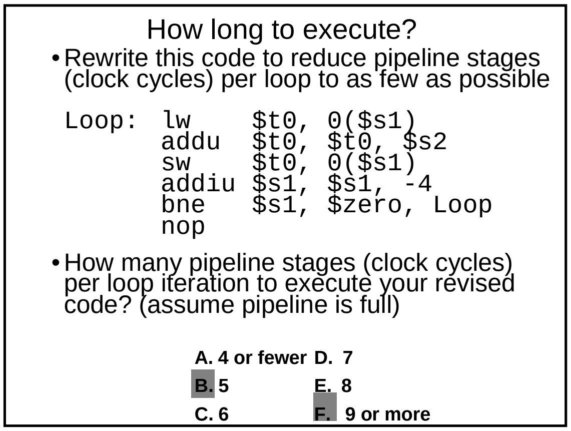
```
                              2. (data hazard so stall)
Loop: 1. lw    $t0, 0($s1)
      3. addu  $t0, $t0, $s2
      4. sw    $t0, 0($s1)
                              7. (data hazard so stall)
      5. addiu $s1, $s1, -4
      6. bne   $s1, $zero, Loop
      8. nop   (delayed branch so exec. nop)
```

- How many pipeline stages (clock cycles) per loop iteration to execute this code?

A. 4 or fewer D. 7

B. 5 [✓ E. 8]

C. 6 F. 9 or more



How long to execute?

- Rewrite this code to reduce pipeline stages (clock cycles) per loop to as few as possible

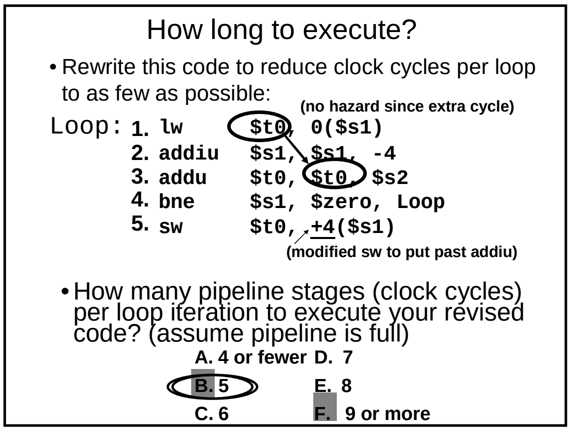
```
Loop: lw    $t0, 0($s1)
      addu  $t0, $t0, $s2
      sw    $t0, 0($s1)
      addiu $s1, $s1, -4
      bne   $s1, $zero, Loop
      nop
```

- How many pipeline stages (clock cycles) per loop iteration to execute your revised code? (assume pipeline is full)

A. 4 or fewer D. 7

B. 5 E. 8

C. 6 F. 9 or more



How long to execute?

- Rewrite this code to reduce clock cycles per loop to as few as possible:

(no hazard since extra cycle)

```
Loop: 1. lw     $t0, 0($s1)
      2. addiu  $s1, $s1, -4
      3. addu   $t0, $t0, $s2
      4. bne    $s1, $zero, Loop
      5. sw     $t0, +4($s1)
```

(modified sw to put past addiu)

- How many pipeline stages (clock cycles) per loop iteration to execute your revised code? (assume pipeline is full)

A. 4 or fewer D. 7

[✓ B. 5] E. 8

C. 6 F. 9 or more