CS61C : Machine Structures

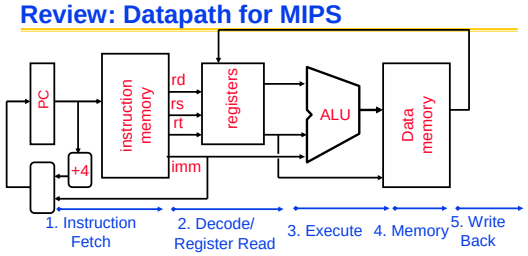
Lecture 6.1.1
Pipelining II

2004-07-26

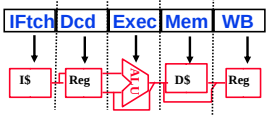
Kurt Meinz

`inst.eecs.berkeley.edu/~cs61c`

Review: Datapath for MIPS



- Use datapath figure to represent pipeline



Review: Problems for Computers

- Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle
  - Structural hazards: HW cannot support this combination of instructions (single person to fold and put clothes away)
  - Control hazards: Pipelining of branches & other instructions stall the pipeline until the hazard; "bubbles" in the pipeline
  - Data hazards: Instruction depends on result of prior instruction still in the pipeline (missing sock)

Review: C.f. Branch Delay vs. Load Delay

- Load Delay occurs only if necessary (dependent instructions).
- Branch Delay always happens (part of the ISA).
- Why not have Branch Delay interlocked?
  - Answer: Interlocks only work if you can detect hazard ahead of time. By the time we detect a branch, we already need its value … hence no interlock is possible!

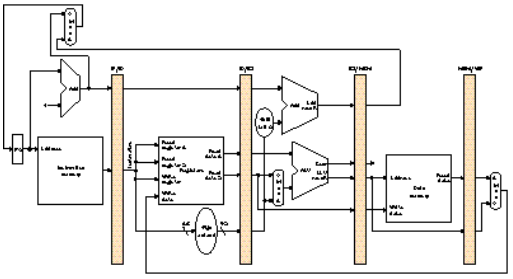
FYI: Historical Trivia

- First MIPS design did not interlock and stall on load-use data hazard
- Real reason for name behind MIPS:
  Microprocessor without
  Interlocked
  Pipeline
  Stages
  - Word Play on acronym for Millions of Instructions Per Second, also called MIPS
  - Load/Use ➔ Wrong Answer!

Outline

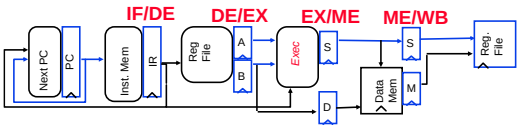
- Pipeline Control
- Forwarding Control
- Hazard Control

Piped Proc So Far …



☹ What's missing??

New Representation: Regs more explicit



IF/DE.Ir = Instruction

DE/EX.A = BusA out of Reg

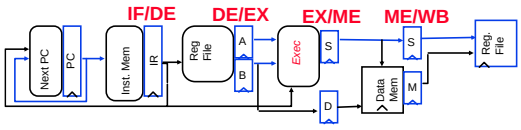
EX/ME.S = AluOut

EX/ME.D = Bus B pass-through for sw

ME/WB.S = ALuOut pass-through

ME/WB.M = Mem Result from lw

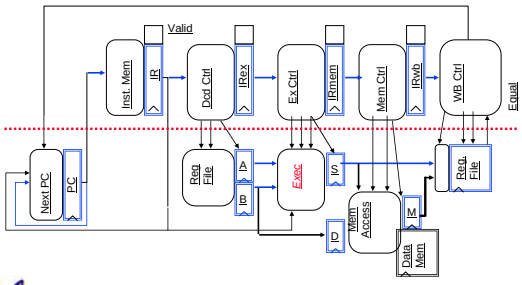
New Representation: Regs more explicit



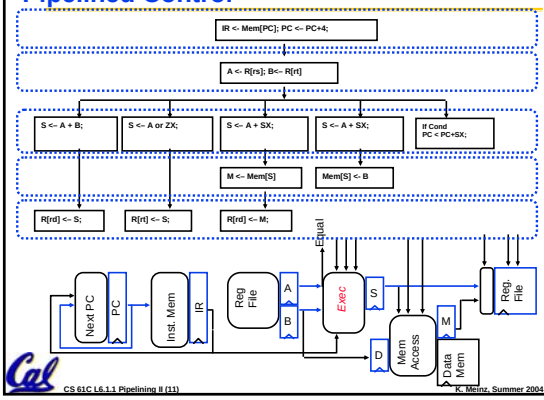
☹ **What's Missing???**

Pipelined Processor (almost) for slides

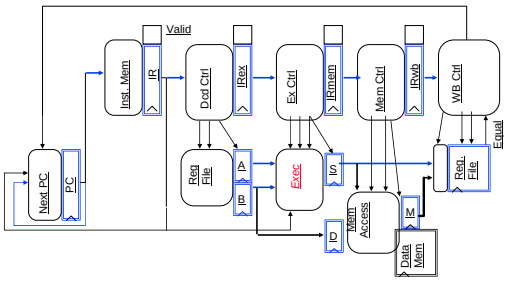
Idea: Parallel Piped Control …



Pipelined Control

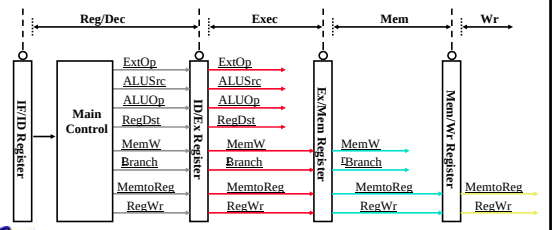


Pipelined Processor



Data Stationary Control

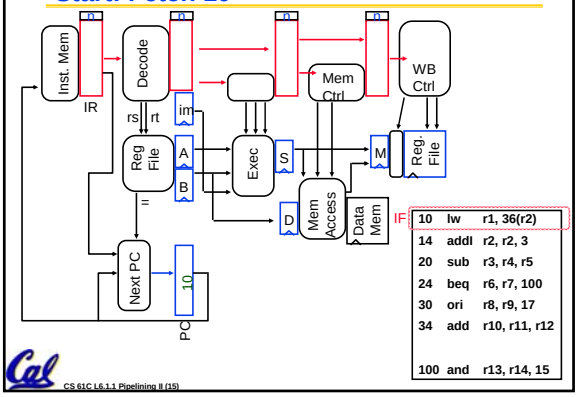
- The Main Control generates the control signals during Reg/Dec
  - Control signals for Exec (ExtOp, ALUSrc, ...) are used 1 cycle later
  - Control signals for Mem (MemWr Branch) are used 2 cycles later
  - Control signals for Wr (MemtoReg MemWr) are used 3 cycles later



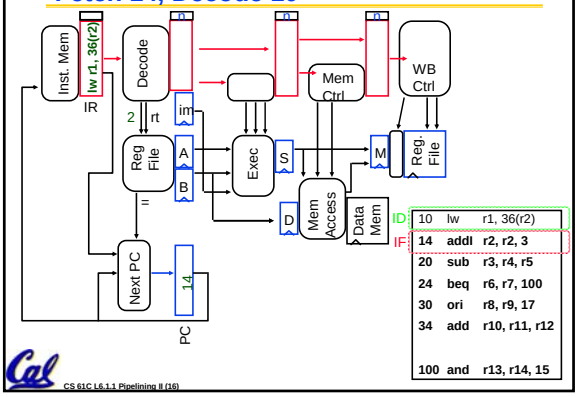
Let's Try it Out

10	lw	r1, 36(r2)
14	addI	r2, r2, 3
20	sub	r3, r4, r5
24	beq	r6, r7, 100
30	ori	r8, r9, 17
34	add	r10, r11, r12
100	and	r13, r14, 15

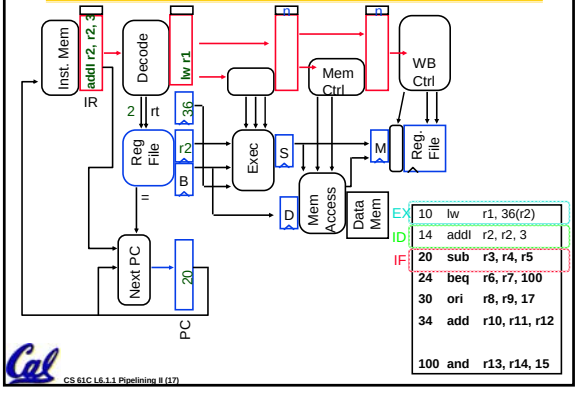
Start: Fetch 10



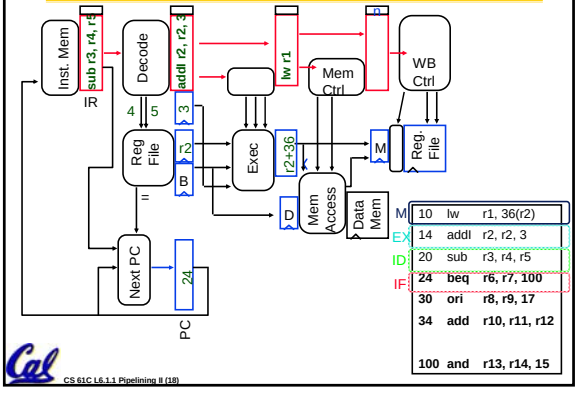
Fetch 14, Decode 10



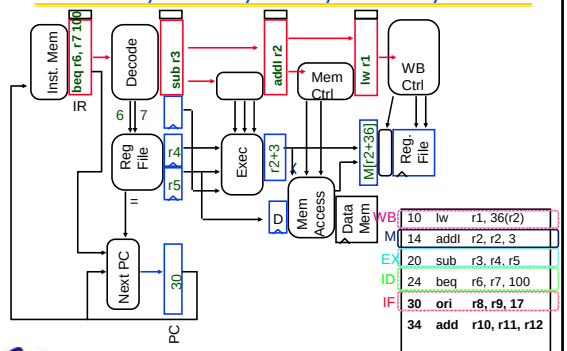
Fetch 20, Decode 14, Exec 10



Fetch 24, Decode 20, Exec 14, Mem 10

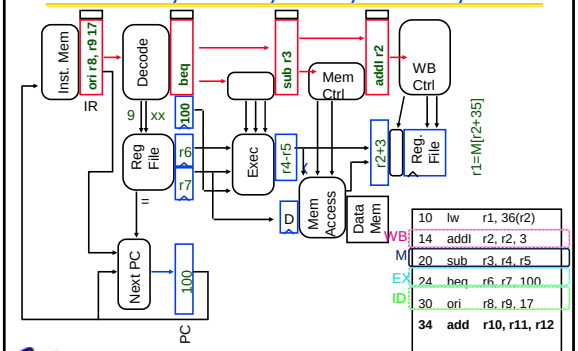


Fetch 30, Dcd 24, Ex 20, Mem 14, WB 10

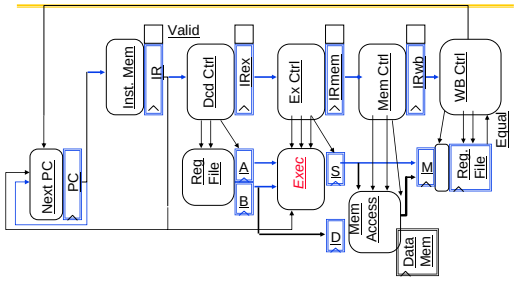


Note Delayed Branch: always execute **`ori`** after **`beq`**

Fetch 100, Dcd 30, Ex 24, Mem 20, WB 14



? ? ? ?



- **Remember: ^ means triggered on edge.**
- **What is wrong here?**

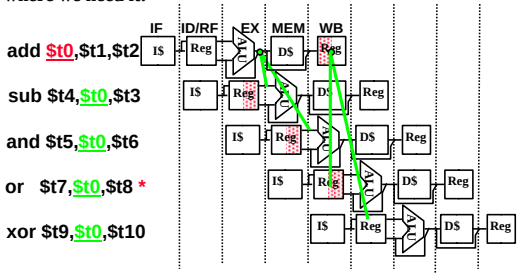
☹.

Outline

- Pipeline Control
- **Forwarding Control**
- **Hazard Control**

Review: Forwarding

Fix by Forwarding result as soon as we have it to where we need it:



*** "or" hazard solved by register hardware**

Forwarding

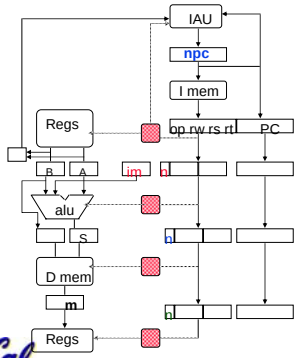
In general:

- For each stage i that has reg inputs
  - For each stage j after I that has reg output
    - If i.reg == j.reg ➔ forward j value back to i.
    - Some exceptions ($0, invalid)

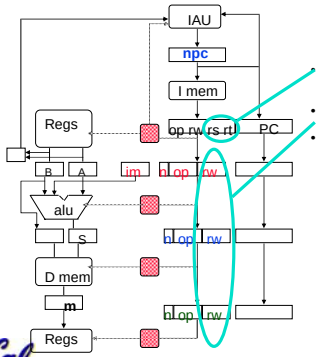
In particular:

- ALUinput ⬅ (ALUResult, MemResult)
- MemInput ⬅ (MemResult)

Pending Writes In Pipeline Registers



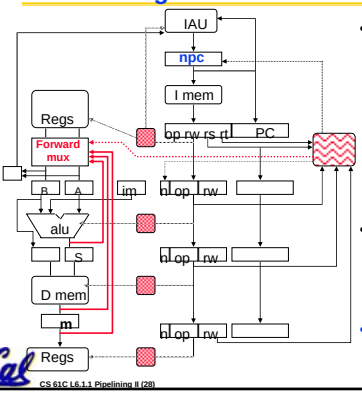
Pending Writes In Pipeline Registers



- Current operand registers
- Pending writes
- hazard <=

  ((rs == rw_{ex}) & $regW_{ex}$) OR
  ((rs == rw_{mem}) & $regW_{me}$) OR
  ((rs == rw_{wb}) & $regW_{wb}$) OR
  ((rt == rw_{ex}) & $regW_{ex}$) OR
  ((rt == rw_{mem}) & $regW_{me}$) OR
  ((rt == rw_{wb}) & $regW_{wb}$)

Forwarding Muxes

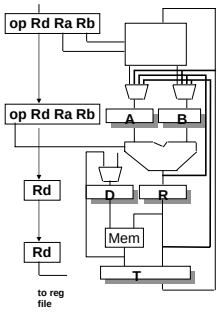


- Detect *nearest* valid write op operand register and forward into op latches, bypassing remainder of the pipe
- Increase muxes to add paths from pipeline registers
- Data Forwarding = Data Bypassing

What about memory operations?

Tricky situation:
MIPS:
lw 0($t0)
sw 0($t1)

RTL:
R1 <- Mem[R2 + I];
Mem[R3+34] <- R1

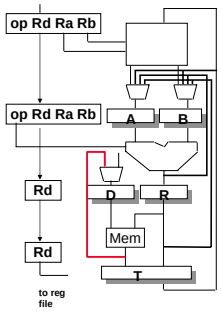


What about memory operations?

Tricky situation:
MIPS:
lw 0($t0)
sw 0($t1)

RTL:
R1 <- Mem[R2 + I];
Mem[R3+34] <- R1

Solution:
Handle with bypass in memory stage!

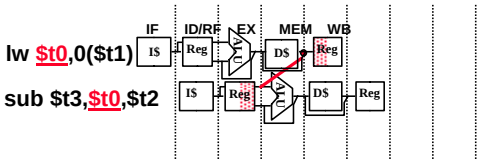


Outline

- Pipeline Control
- Forwarding Control
- Hazard Control

Data Hazard: Loads (1/4)

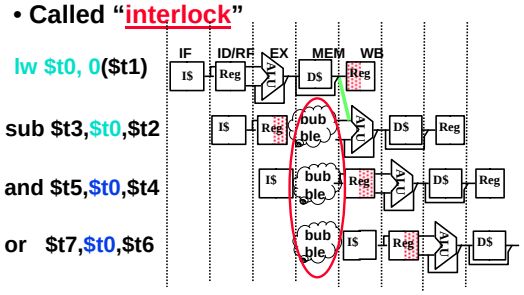
- Forwarding works if value is available (but not written back) before it is needed. But consider …



- Need result before it is calculated!
- Must stall use (sub) 1 cycle and *then* forward. …

Data Hazard: Loads (2/4)

- Hardware must stall pipeline
- Called "interlock"

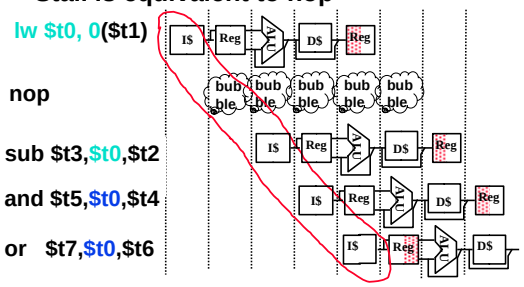


Data Hazard: Loads (3/4)

- 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



Hazards / Stalling

In general:

- For each stage i that has reg inputs
  - If I's reg is being written later on in the pipe but is not ready yet
    - Stages 0 to i: Stall (Turn CEs off so no change)
    - Stage i+1: Make a bubble (do nothing)
    - Stages i+2 onward: As usual

In particular:

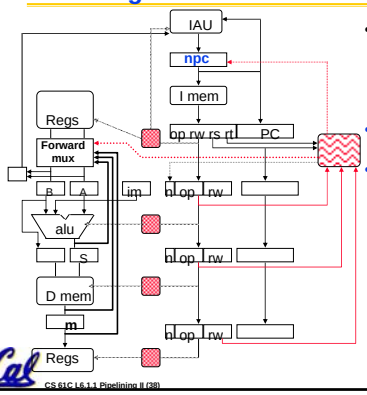
- ALUinput ← (MemResult)

Hazards / Stalling

Alternative Approach:

- Detect non-forwarding hazards in decode
  - Possible since our hazards are *formal*.
    - Not always the case.
  - Stalling then becomes:
    - Issue nop to EX stage
    - Turn off nextPC update (refetch same inst)
    - Turn off InstReg update (re-decode same inst)

Stall Logic



- 1. Detect non-resolving hazards.
- 2a. Insert Bubble
- 2b. Stall nextPC, IF/DE

Stall Logic

- Stall-on-issue is used quite a bit
  - More complex processors: many cases that stall on issue.
  - More complex processors: cases that can't be detected at decode
    - E.g. value needed from mem is not in cache – proc must stall multiple cycles

By the way …

- Notice that our forwarding and stall logic is stateless!
- Big Idea: Keep it simple!
  - Option 1: Store old fetched inst in reg ("stall_temp"), keep state reg that says whether to use stall_temp or value coming off inst mem.
  - Option 2: Re-fetch old value by turning off PC update.