COMP3211 03S2 Lecture 23

Introduction to Pipelining

Adapted from

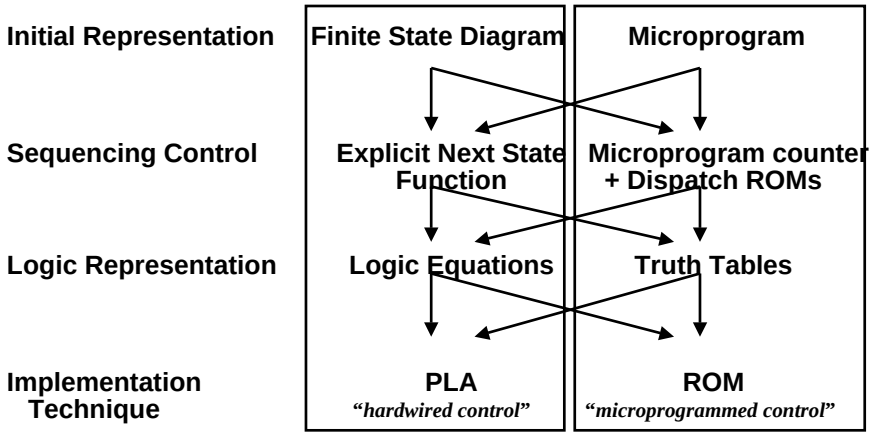
CS152: Computer Architecture and Engineering
Dave Patterson (www.cs.berkeley.edu/~pattrsn)

Recap last week

- Disadvantages of the Single Cycle Processor
  - Long cycle time
  - Cycle time is too long for all instructions except the Load
- 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
- Control is specified by finite state diagram
- Specialized state-diagrams easily captured by microsequencer
  - simple increment & "branch" fields
  - datapath control fields

Overview of Control

- Control may be designed using one of several initial representations. The choice of sequence control, and how logic is represented, can then be determined independently; the control can then be implemented with one of several methods using a structured logic technique.



Microprogramming Pros and Cons

- Ease of design
- Flexibility
  - Easy to adapt to changes in organization, timing, technology
  - Can make changes late in design cycle, or even in the field
- Can implement very powerful instruction sets (just more control memory)
- Generality
  - Can implement multiple instruction sets on same machine.
  - Can tailor instruction set to application.
- Compatibility
  - Many organizations, same instruction set
- Slow
- Costly to implement

Summary: Microprogramming one inspiration for RISC

° If simple instruction could execute at very high clock rate…

° If you could even write compilers to produce microinstructions…

° If most programs use simple instructions and addressing modes…

° If microcode is kept in RAM instead of ROM so as to fix bugs …

° If same memory used for control memory could be used instead as cache for "macroinstructions"…

° Then why not skip instruction interpretation by a microprogram and simply compile directly into lowest language of machine? (microprogramming is overkill when ISA matches datapath 1-1)

Summary (cont'd)

° Microprogramming is a fundamental concept
- implement an instruction set by building a very simple processor and interpreting the instructions
- essential for very complex instructions and when few register transfers are possible
- overkill when ISA matches datapath 1-1

° Control is more complicated with:
- complex instruction sets
- restricted datapaths (e.g. single memory)

° Simple Instruction set and powerful datapath => simple control
- could try to reduce hardware (see the book)
- rather go for speed => many instructions at once!

Pipelining is Natural!

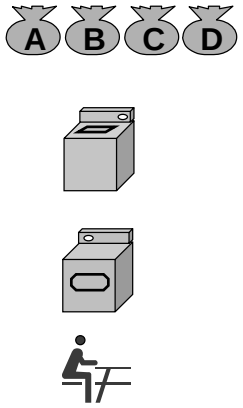
° Laundry Example

° Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold

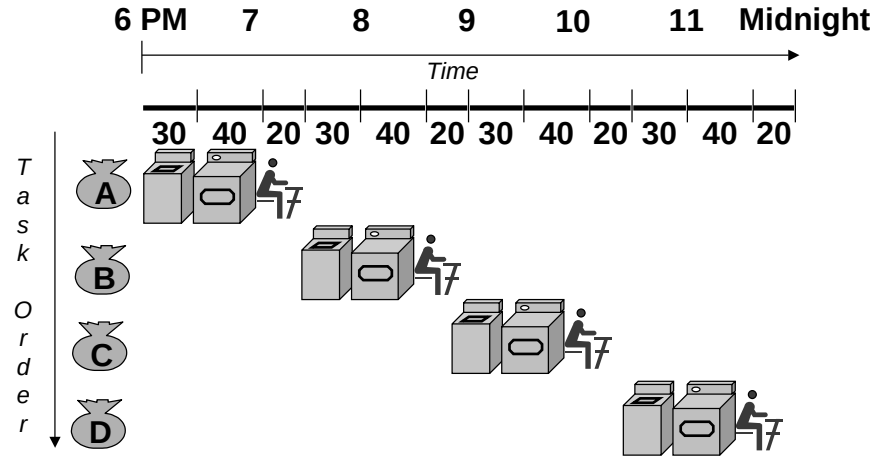
° Washer takes 30 minutes

° Dryer takes 40 minutes

° "Folder" takes 20 minutes



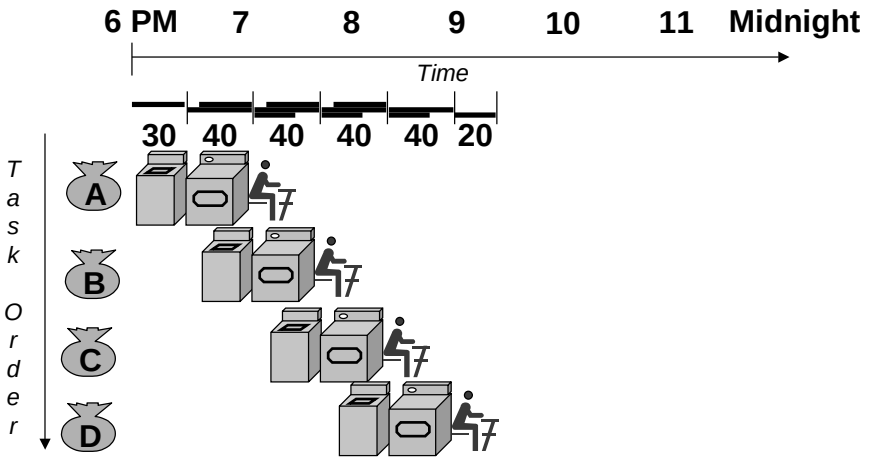
Sequential Laundry



° Sequential laundry takes 6 hours for 4 loads

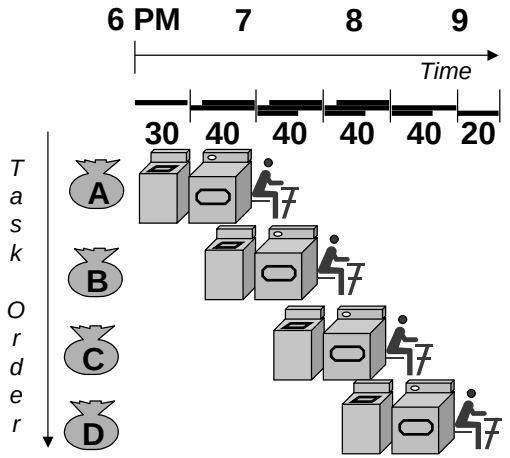
° If they learned pipelining, how long would laundry take?

Pipelined Laundry: Start work ASAP



° Pipelined laundry takes 3.5 hours for 4 loads

Pipelining Lessons



° Pipelining doesn't help latency of single task, it helps throughput of entire workload

° Pipeline rate limited by slowest pipeline stage

° Multiple tasks operating simultaneously using different resources

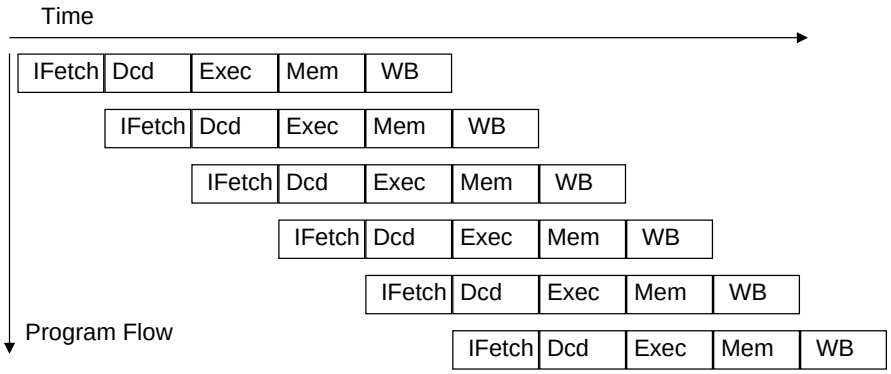
° Potential speedup = Number pipe stages

° Unbalanced lengths of pipe stages reduces speedup

° Time to "fill" pipeline and time to "drain" it reduces speedup

° Stall for Dependences

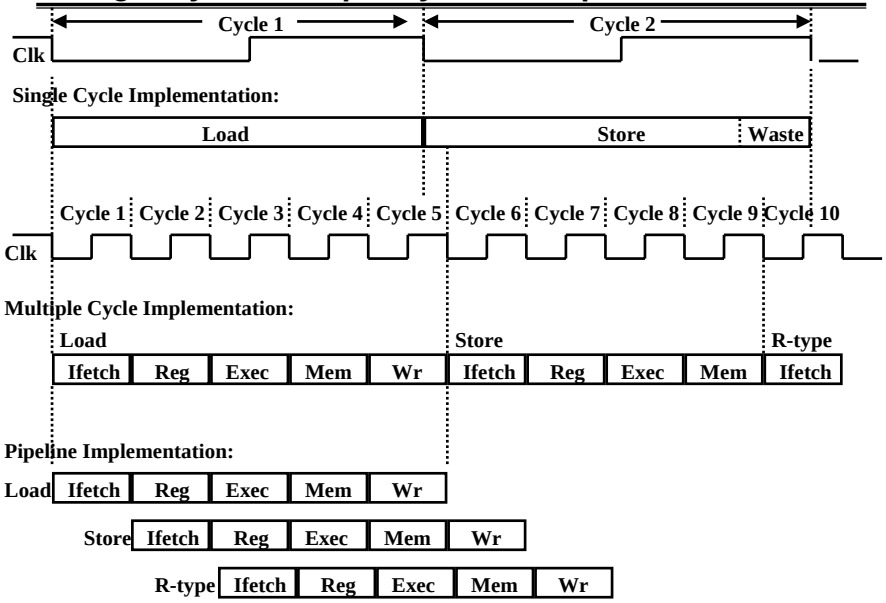
Pipelined Processor Execution



° Utilization?

° Now we just have to make it work

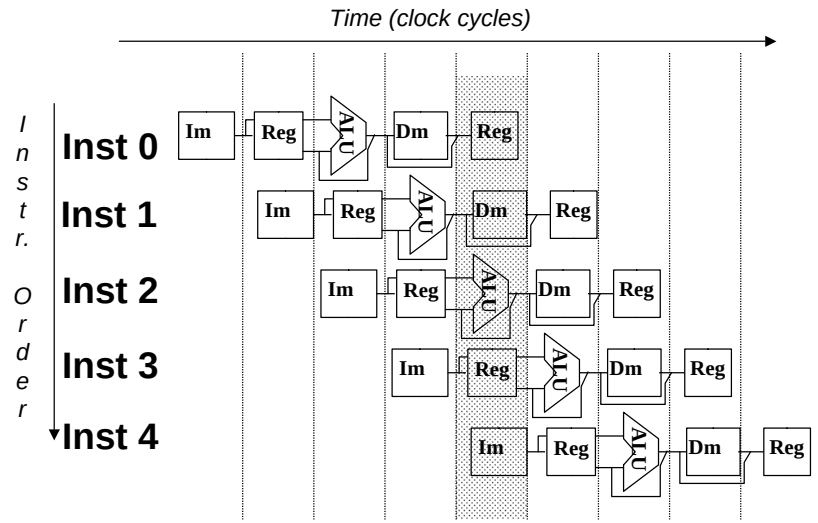
Single Cycle, Multiple Cycle, vs. Pipeline



Why Pipeline?

- Suppose we execute 100 instructions
- Single Cycle Machine
  - 45 ns/cycle x 1 CPI x 100 inst = 4500 ns
- Multicycle Machine
  - 10 ns/cycle x 4.1 CPI (due to inst mix) x 100 inst = 4100 ns
- Ideal pipelined machine
  - 10 ns/cycle x (1 CPI x 100 inst + 4 cycle drain) = 1040 ns

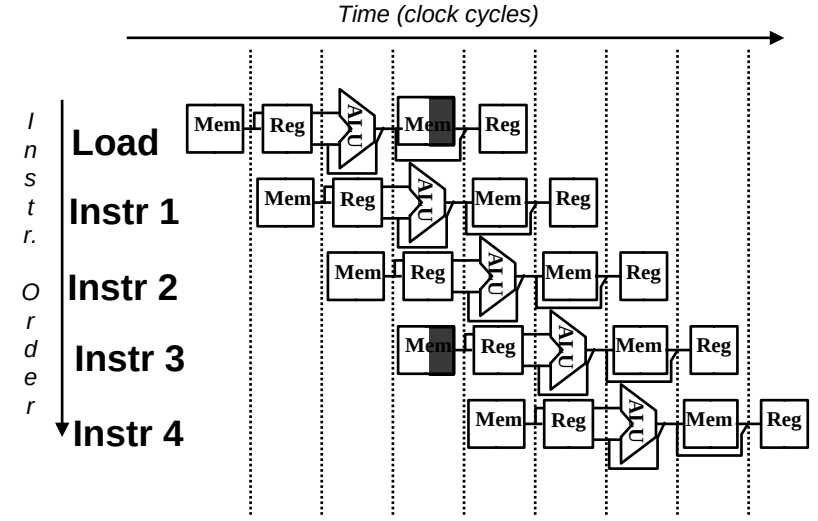
Why Pipeline? Because the resources are there!



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 doing something else (watching TV)
    - single memory for instructions and data cannot be accessed simultaneously
  - 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
  - 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
- 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)

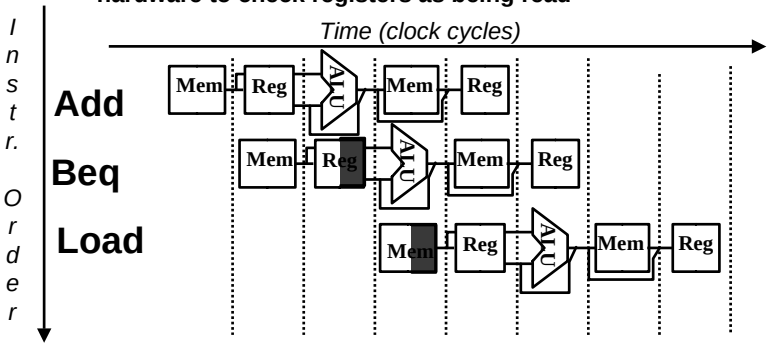
Structural Hazards limit performance

° Example: if 1.3 memory accesses per instruction and only one memory access per cycle then
  - average CPI ≥ 1.3
  - otherwise resource is more than 100% utilized

° Solution is to provide two memories

Control Hazard Solutions

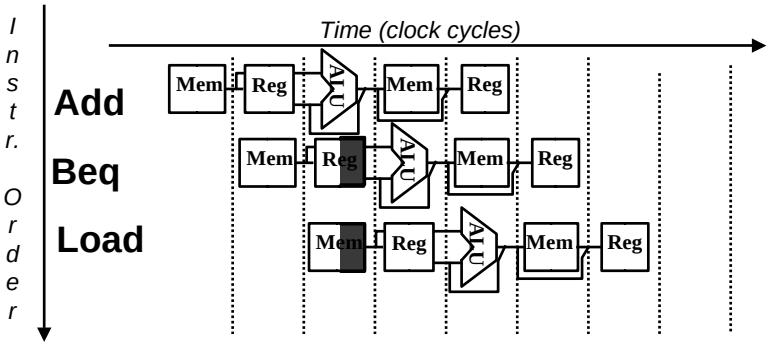
° Stall: wait until decision is clear
  - Its possible to move up decision to 2nd stage by adding hardware to check registers as being read



° Impact: 2 clock cycles per branch instruction => slow

Control Hazard Solutions

° Predict: guess one direction then back up if wrong
  - Predict not taken

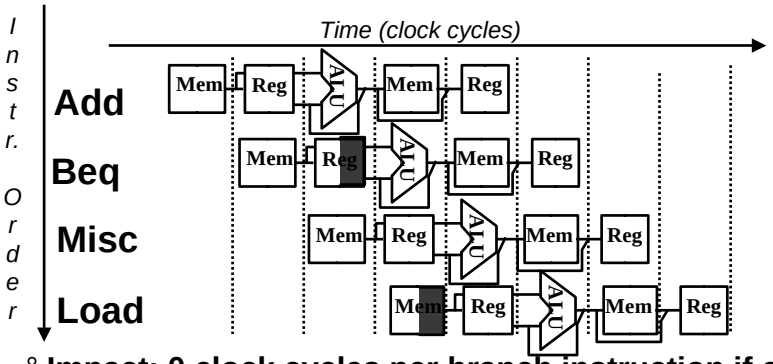


° Impact: 1 clock cycle per branch instruction if right, 2 if wrong (right - 50% of time)

° More dynamic scheme: history of 1 branch (~90%√)

Control Hazard Solutions

° Redefine branch behavior (takes place after next instruction) “delayed branch”



° 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 Hazard on r1

add r1 ,r2,r3

sub r4, r1 ,r3

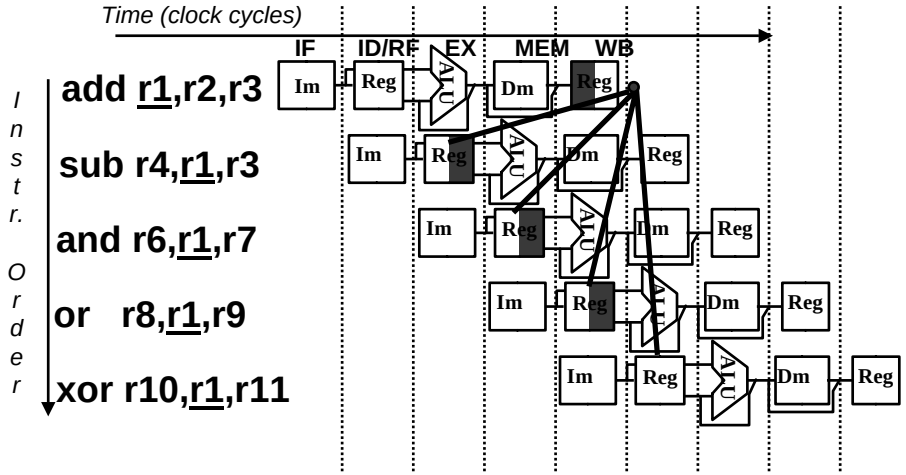
and r6, r1 ,r7

or r8, r1 ,r9

xor r10, r1 ,r11

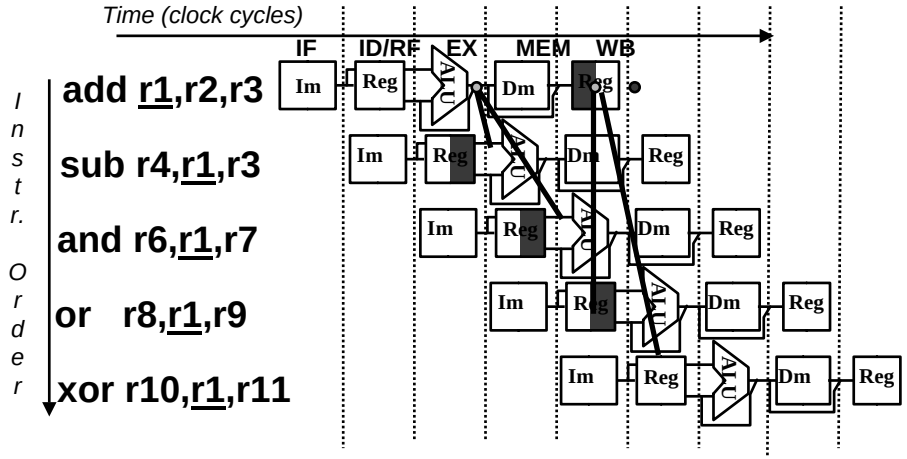
Data Hazard on r1:

- Dependencies backwards in time are hazards



Data Hazard Solution:

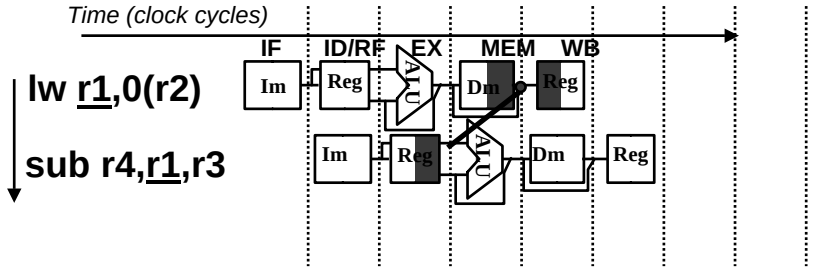
- "Forward" result from one stage to another



- "or" OK if define read/write properly

Forwarding (or Bypassing): What about Loads

° Dependencies backwards in time are hazards



° Can't solve with forwarding:

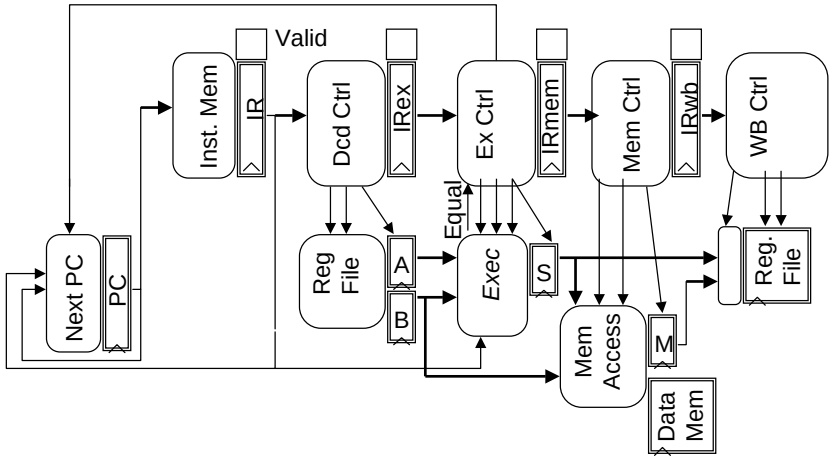
- Must delay/stall instruction dependent on loads

Designing a Pipelined Processor

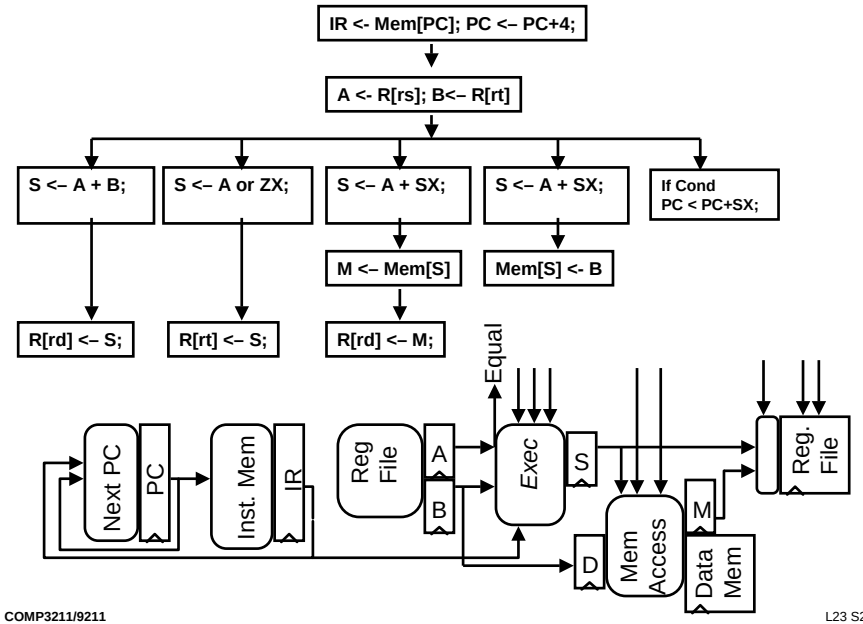
1. Examine datapath and control diagram
2. Determine resources associated with states
3. Ensure that flows do not conflict, or figure out how to resolve
4. Assert control at appropriate stage

Pipelined Processor (almost same as book) for slides

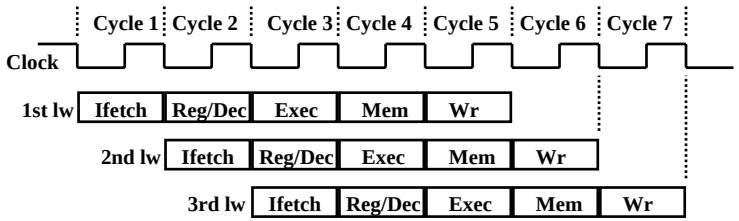
° What happens if we start a new instruction every cycle?



Control and Datapath



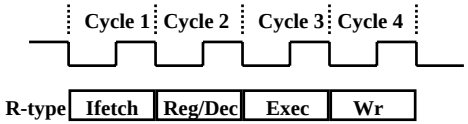
Pipelining the Load Instruction



° The five independent functional units in the pipeline datapath are:

- Instruction Memory for the Ifetch stage
- Register File's Read ports (bus A and busB) for the Reg/Dec stage
- ALU for the Exec stage
- Data Memory for the Mem stage
- Register File's Write port (bus W) for the Wr stage

The Four Stages of R-type



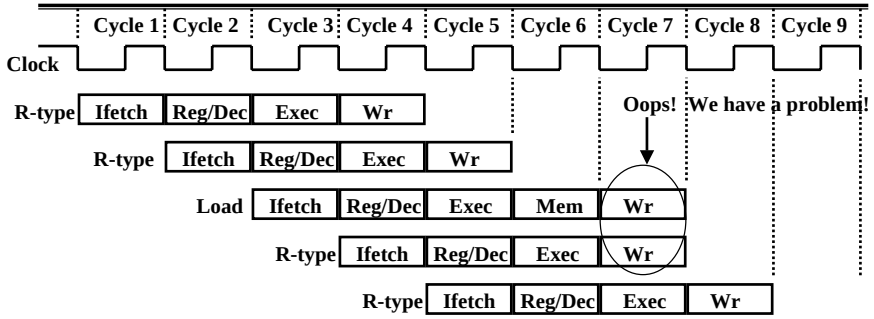
° **Ifetch: Instruction Fetch**
  - Fetch the instruction from the Instruction Memory

° **Reg/Dec: Registers Fetch and Instruction Decode**

° **Exec:**
  - ALU operates on the two register operands
  - Update PC

° **Wr: Write the ALU output back to the register file**

Pipelining the R-type and Load Instruction

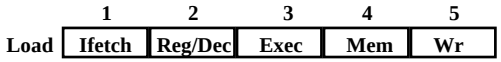


° **We have pipeline conflict or structural hazard:**
  - Two instructions try to write to the register file at the same time!
  - Only one write port

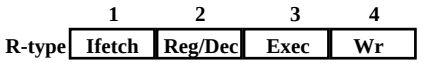
Important Observation

° **Each functional unit can only be used once per instruction**

° **Each functional unit must be used at the same stage for all instructions:**
  - Load uses Register File's Write Port during its 5th stage

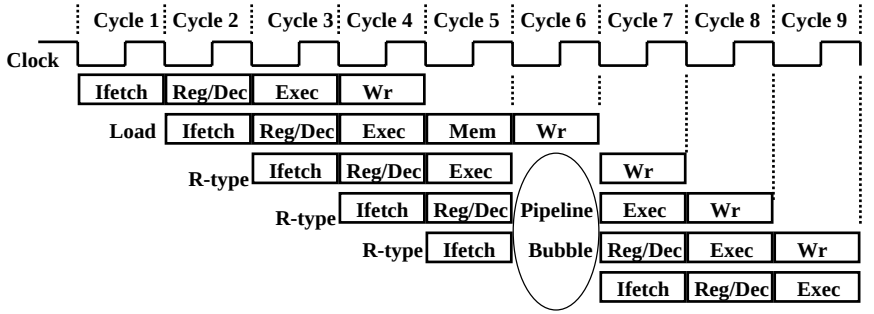


  - R-type uses Register File's Write Port during its 4th stage



° **2 ways to solve this pipeline hazard.**

Solution 1: Insert "Bubble" into the Pipeline

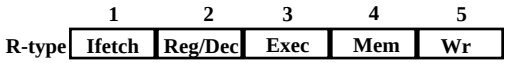


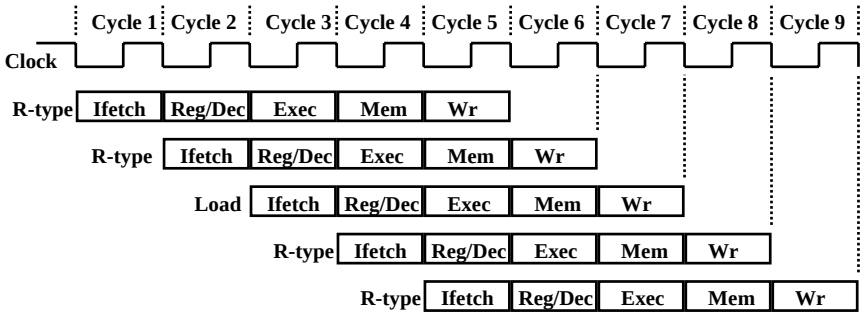
° **Insert a "bubble" into the pipeline to prevent 2 writes at the same cycle – stalls subsequent instructions at the stage they have reached so far**
  - The control logic can be complex.
  - Lose instruction fetch and issue opportunity.

° **No instruction is started in Cycle 6!**

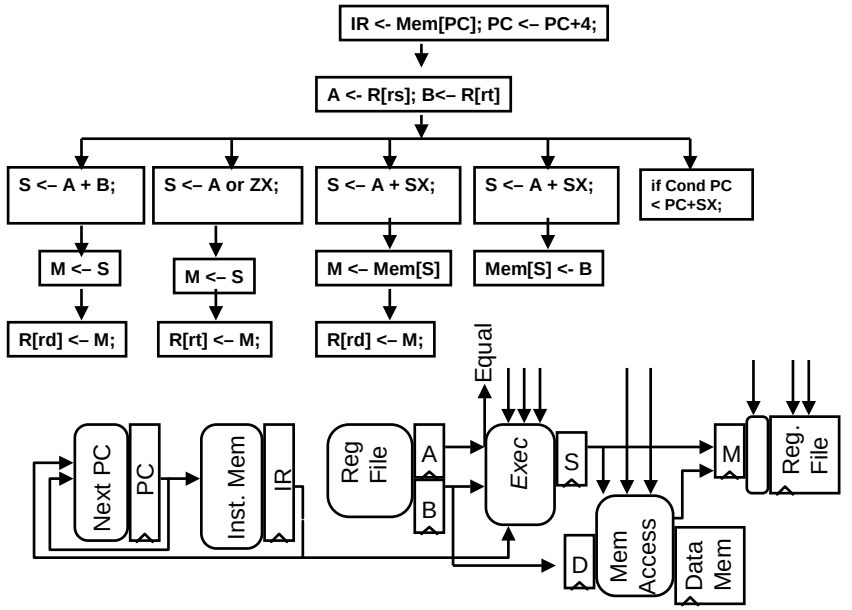
Solution 2: Delay R-type's Write by One Cycle

° Delay R-type's register write by one cycle:
- Now R-type instructions also use Reg File's write port at Stage 5
- Mem stage is a NOOP stage: nothing is being done.

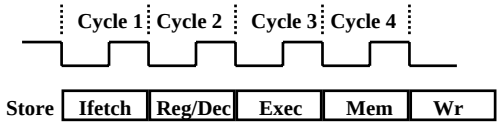




Modified Control & Datapath



The Four Stages of Store



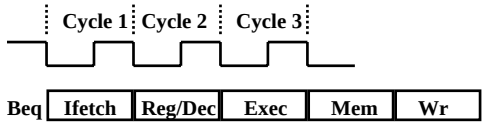
° Ifetch: Instruction Fetch
- Fetch the instruction from the Instruction Memory

° Reg/Dec: Registers Fetch and Instruction Decode

° Exec: Calculate the memory address

° Mem: Write the data into the Data Memory

The Three Stages of Beq

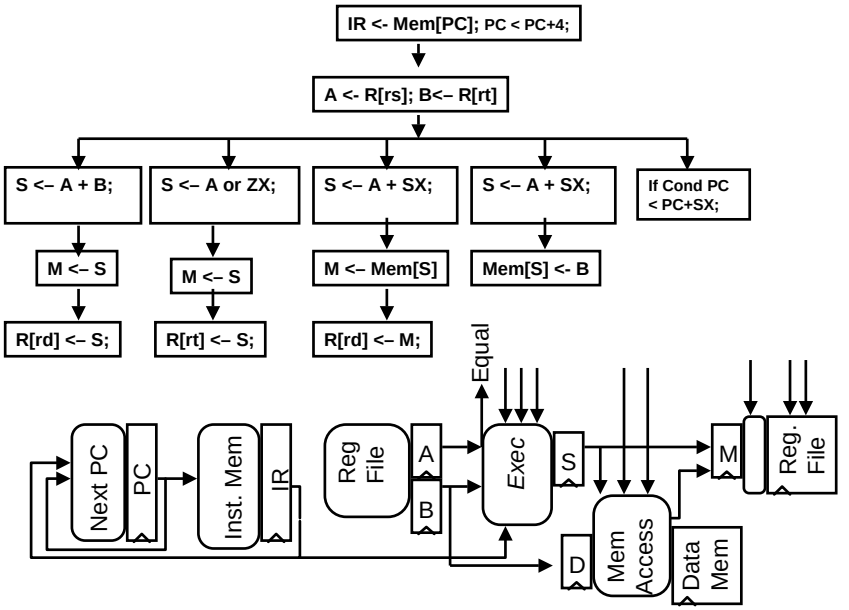


° Ifetch: Instruction Fetch
- Fetch the instruction from the Instruction Memory

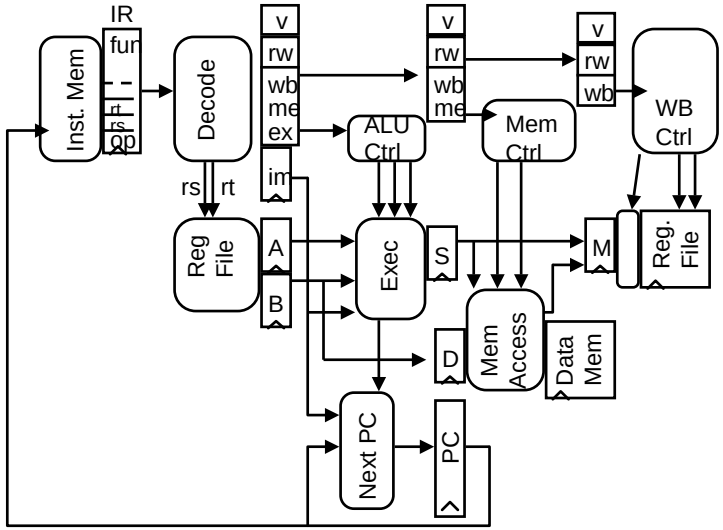
° Reg/Dec:
- Registers Fetch and Instruction Decode

° Exec:
- compares the two register operand,
- select correct branch target address
- latch into PC

Control Diagram



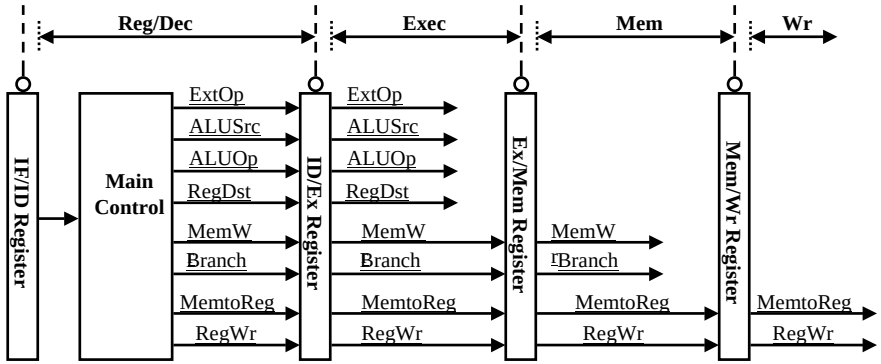
Datapath + Data Stationary Control



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**

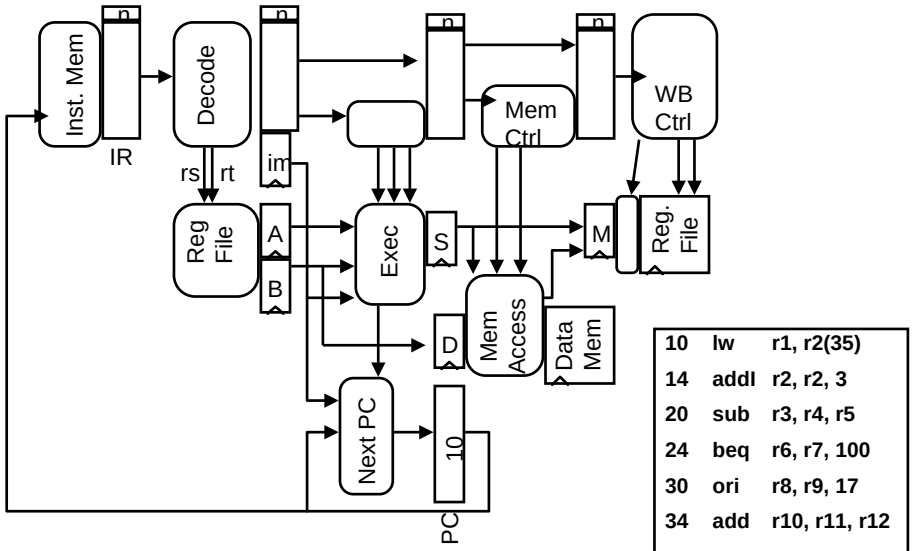


Let's Try it Out

10	**lw**	**r1, r2(35)**
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**

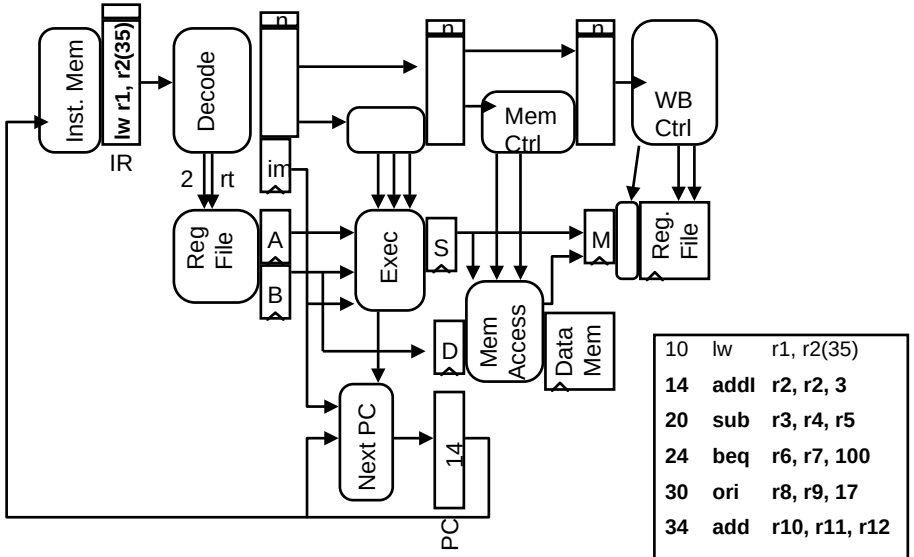
these addresses are octal

Start: Fetch 10



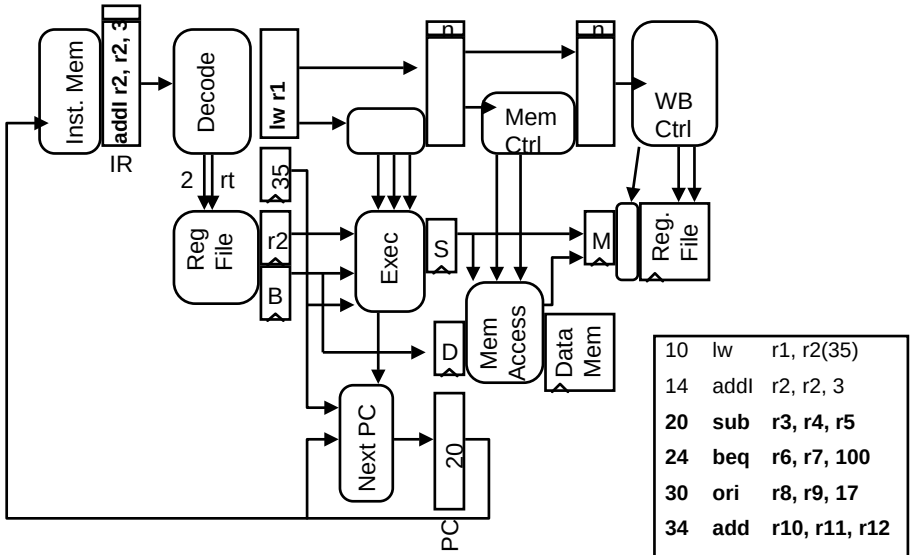
10	**lw**	**r1, r2(35)**
14	**addl**	**r2, r2, 3**
20	**sub**	**r3, r4, r5**
24	**beq**	**r6, r7, 100**
30	**ori**	**r8, r9, 17**
34	**add**	**r10, r11, r12**
100	**and**	**r13, r14, 15**

Fetch 14, Decode 10



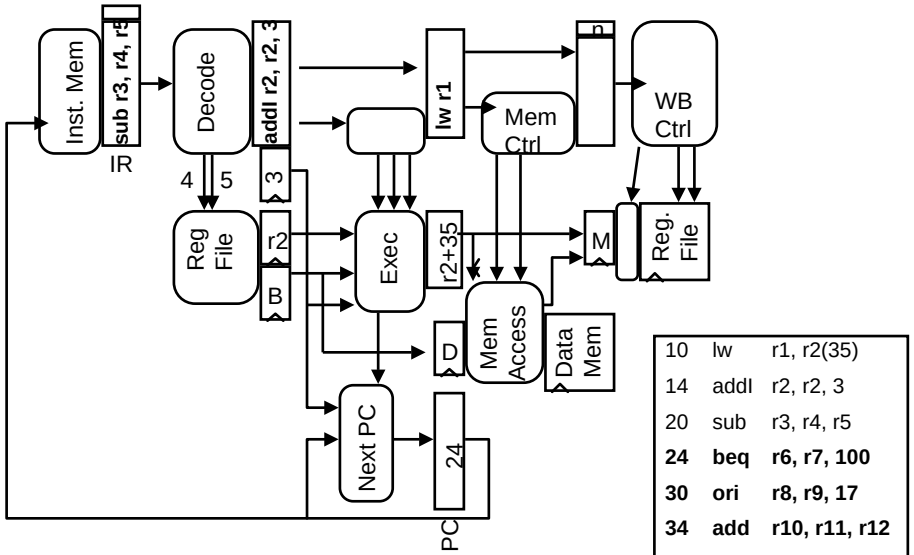
10	lw	r1, r2(35)
14	**addl**	**r2, r2, 3**
20	**sub**	**r3, r4, r5**
24	**beq**	**r6, r7, 100**
30	**ori**	**r8, r9, 17**
34	**add**	**r10, r11, r12**
100	**and**	**r13, r14, 15**

Fetch 20, Decode 14, Exec 10



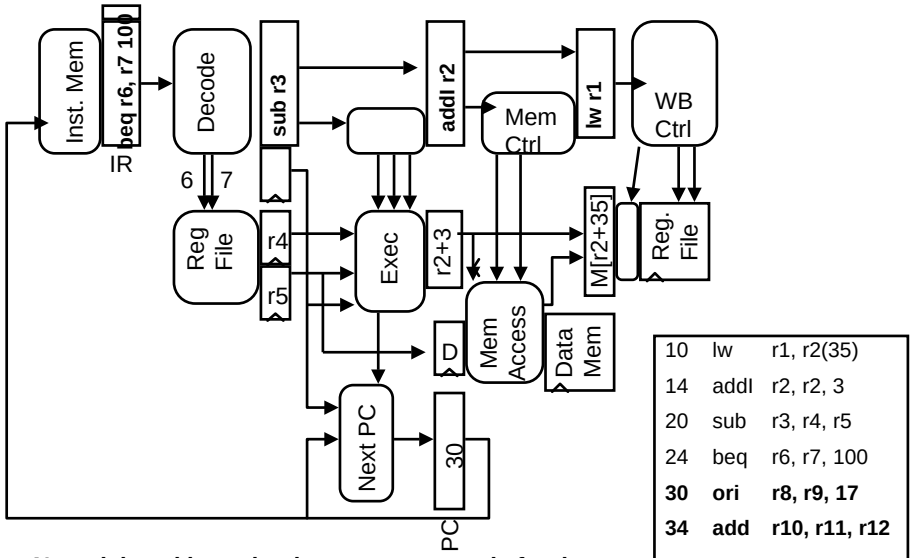
10	lw	r1, r2(35)
14	addl	r2, r2, 3
20	**sub**	**r3, r4, r5**
24	**beq**	**r6, r7, 100**
30	**ori**	**r8, r9, 17**
34	**add**	**r10, r11, r12**
100	**and**	**r13, r14, 15**

Fetch 24, Decode 20, Exec 14, Mem 10



10	lw	r1, r2(35)
14	addl	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**

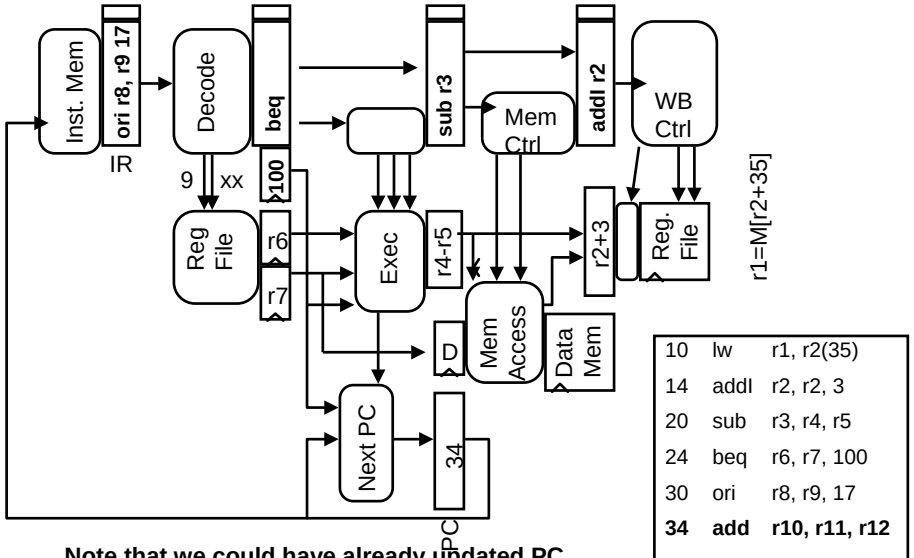
Fetch 30, Dcd 24, Ex 20, Mem 14, WB 10



10	lw	r1, r2(35)
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**

Note delayed branch: always execute ori after beq

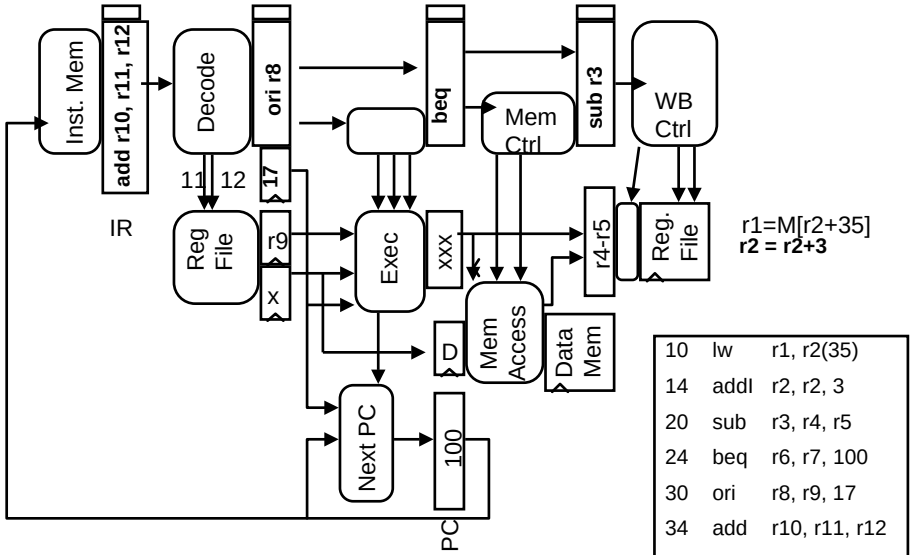
Fetch 34, Dcd 30, Ex 24, Mem 20, WB 14



10	lw	r1, r2(35)
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**

Note that we could have already updated PC
if Reg File compared r6 & r7 as they were retrieved

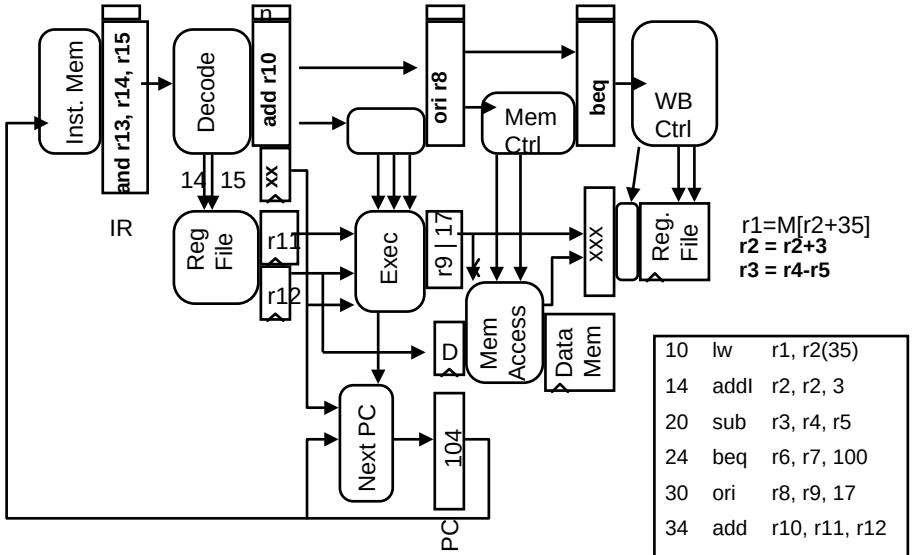
Fetch 100, Dcd 34, Ex 30, Mem 24, WB 20



10	lw	r1, r2(35)
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**

ooops, we should have only one delayed instruction

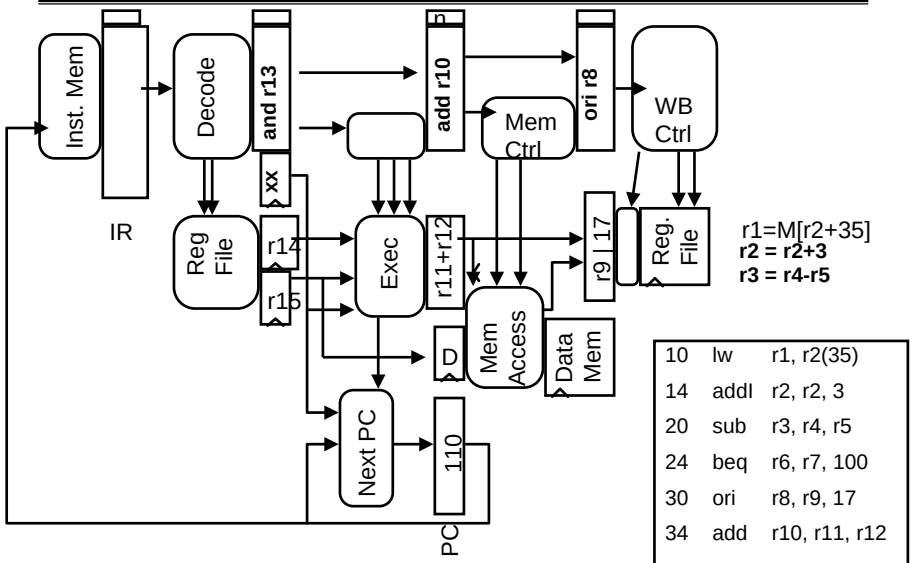
Fetch 104, Dcd 100, Ex 34, Mem 30, WB 24



10	lw	r1, r2(35)
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**

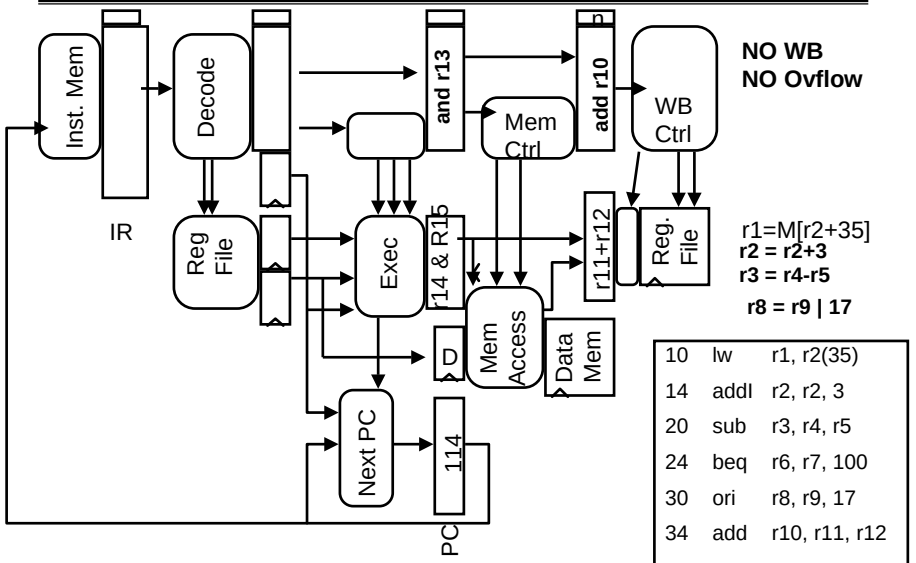
Squash the extra instruction in the branch shadow!

Fetch 108, Dcd 104, Ex 100, Mem 34, WB 30



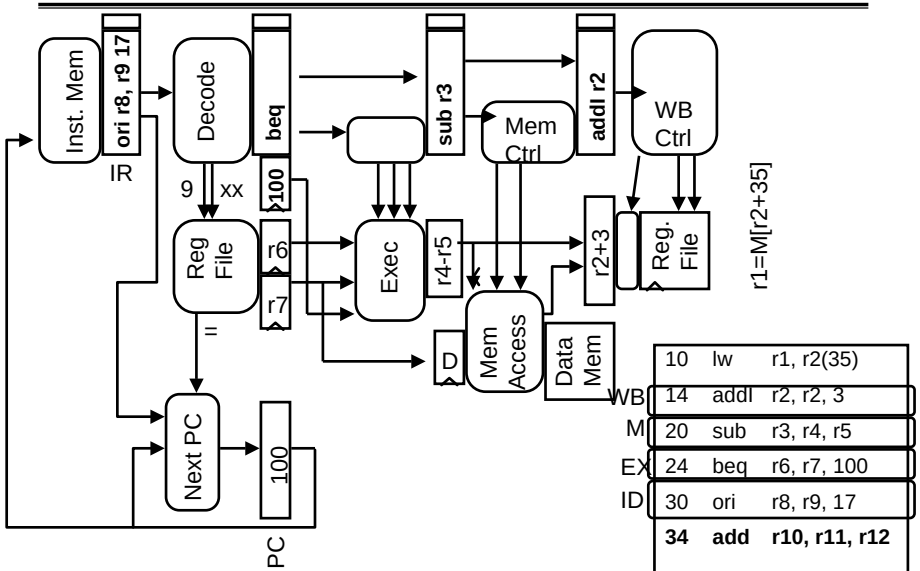
10	lw	r1, r2(35)
14	addI	r2, r2, 3
20	sub	r3, r4, r5
24	beq	r6, r7, 100
30	ori	r8, r9, 17
34	add	r10, r11, r12
100	**and**	**r13, r14, 15**

Fetch 114, Dcd 110, Ex 104, Mem 100, WB 34



10	lw	r1, r2(35)
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**

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



10	lw	r1, r2(35)
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**

Early update of PC when Reg file compares contents

Summary: Pipelining

° **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**

Summary

- **Pipelining is a fundamental concept**
  - **multiple steps using distinct resources**
- **Utilize capabilities of the Datapath by pipelined instruction processing**
  - **start next instruction while working on the current one**
  - **limited by length of longest stage (plus fill/flush)**
  - **detect and resolve hazards**