Computer System Architecture

Pipelining Part II

Chalermek Intanagonwiwat

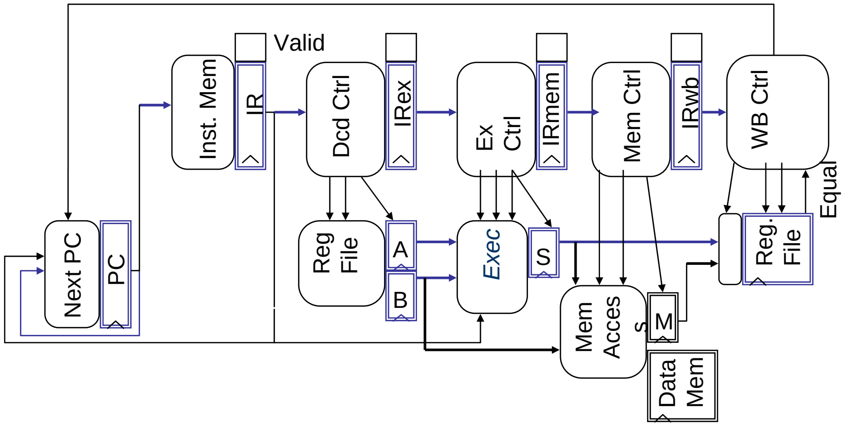
Slides courtesy of David Patterson

Designing a Pipelined Processor

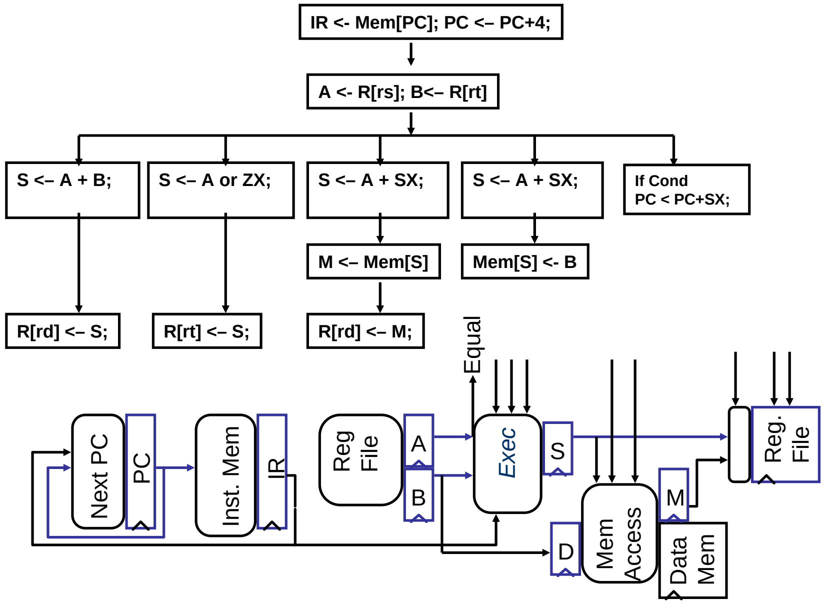
- Go back and examine your datapath and control diagram
- associated resources with states
- ensure that flows do not conflict, or figure out how to resolve
- assert control in appropriate stage

Pipelined Processor

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



Control and Datapath


IR <- Mem[PC]; PC <– PC+4;

A <- R[rs]; B<– R[rt]

S <– A + B;

S <– A or ZX;

S <– A + SX;

S <– A + SX;

If Cond PC < PC+SX;

M <– Mem[S]

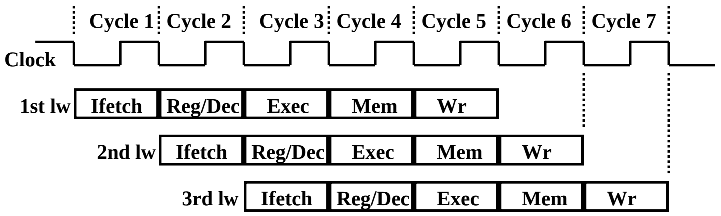
Mem[S] <- B

R[rd] <– S;

R[rt] <– S;

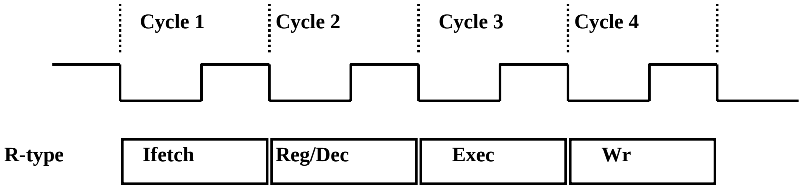
R[rd] <– M;

Pipelining the Load Instruction



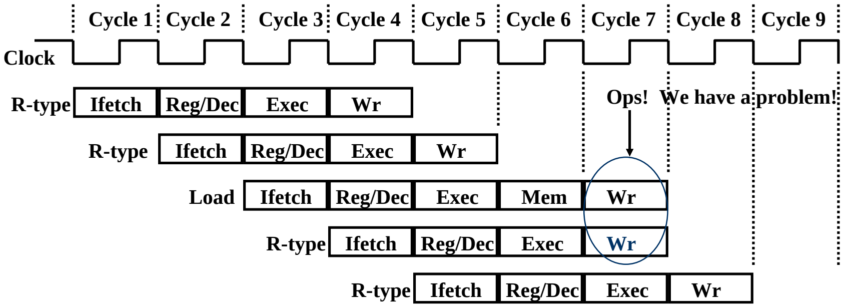
- The five independent functional units in the pipeline datapath are:
  - Instruction Memory for the Ifetch stage
  - Register File's Read ports (bus A and busB) for the Reg/Dec stage
  - ALU for the Exec stage
  - Data Memory for the Mem stage
  - Register File's Write port (bus W) for the Wr stage

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

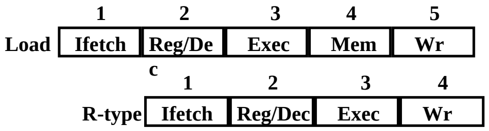


- We have pipeline conflict or structural hazard:
  - Two instructions try to write to the register file at the same time!
  - Only one write port

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

	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle 9
Clock									
	Ifetch	Reg/Dec	Exec	Wr					
Load		Ifetch	Reg/Dec	Exec	Mem	Wr			
R-type			Ifetch	Reg/Dec	Exec		Wr		
R-type				Ifetch	Reg/Dec	Pipeline	Exec	Wr	
R-type					Ifetch	Bubble	Reg/Dec	Exec	Wr
							Ifetch	Reg/Dec	Exec

- Insert a "bubble" into the pipeline to prevent 2 writes at the same cycle
  - 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

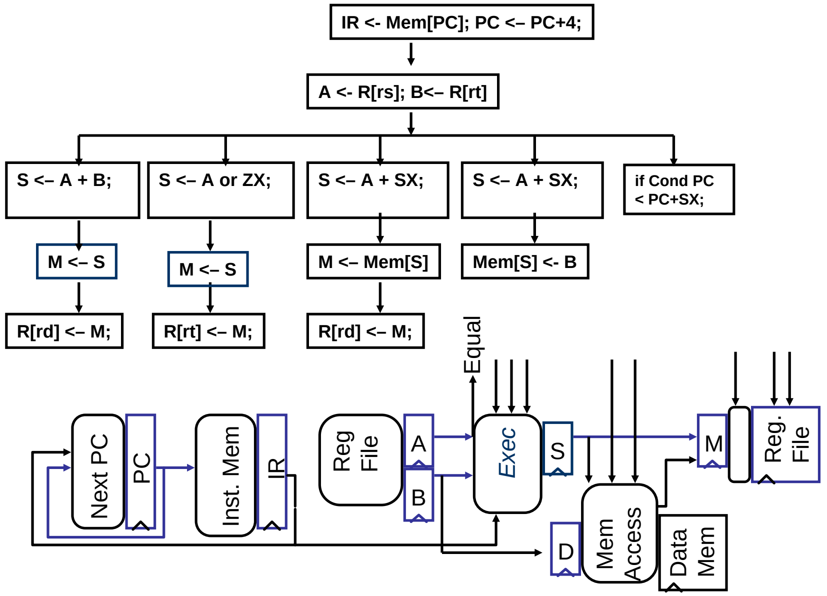
- Delay R-type's register write by one cycle:
  - Now R-type instructions also use Reg File's write port at Stage 5
  - Mem stage is a NOOP stage: nothing is being done.

	1	2	3	4	5
R-type	Ifetch	Reg/Dec	Exec	Mem	Wr

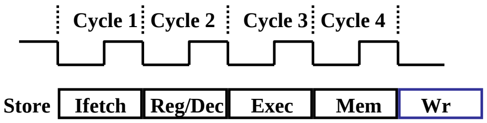
Solution 2: Delay R-type's Write by One Cycle (cont.)

	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle 9
Clock									
R-type	Ifetch	Reg/Dec	Exec	Mem	Wr				
R-type		Ifetch	Reg/Dec	Exec	Mem	Wr			
Load			Ifetch	Reg/Dec	Exec	Mem	Wr		
R-type				Ifetch	Reg/Dec	Exec	Mem	Wr	
R-type					Ifetch	Reg/Dec	Exec	Mem	Wr

Modified Control & Datapath

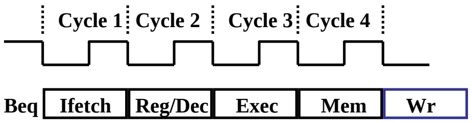


The Four Stages of Store



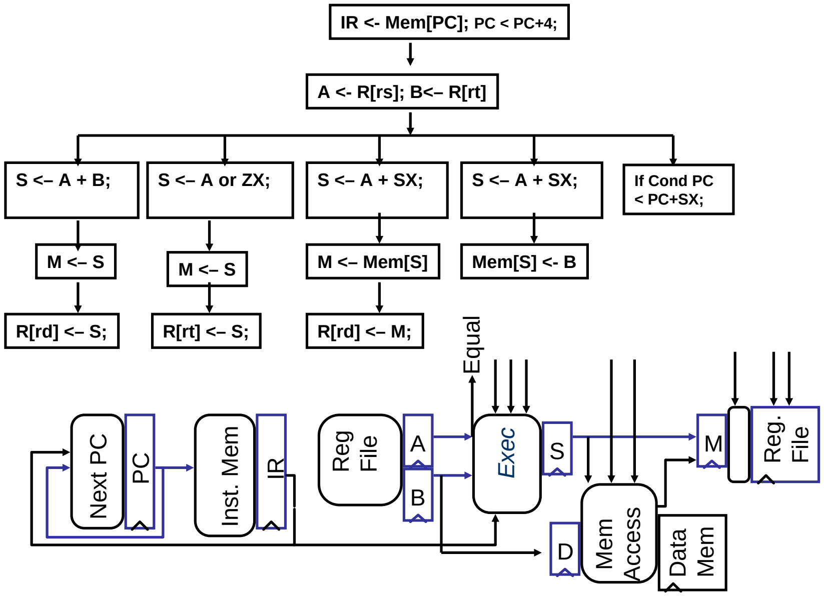
- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec: Calculate the memory address
- Mem: Write the data into the Data Memory

The Three Stages of Beq

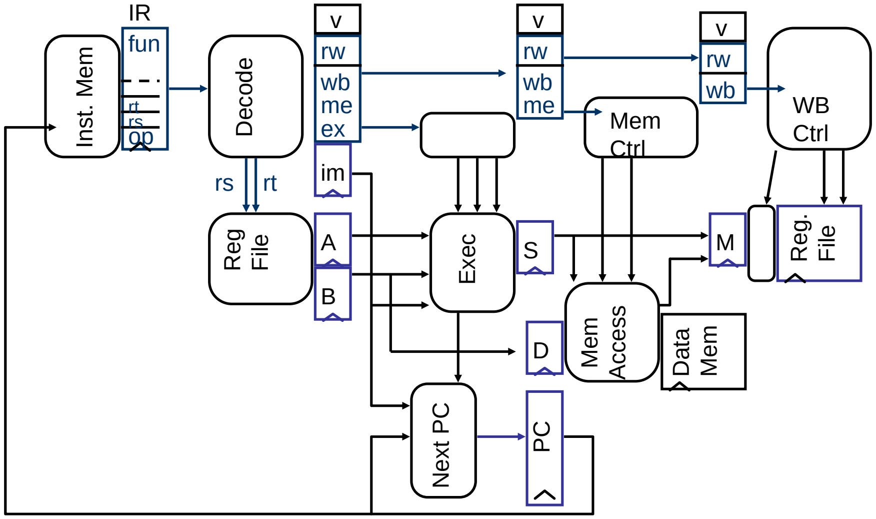


- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- Reg/Dec:
  - Registers Fetch and Instruction Decode
- Exec:
  - compares the two register operand,
  - select correct branch target address
  - latch into PC

Control Diagram



Datapath + Data Stationary Control

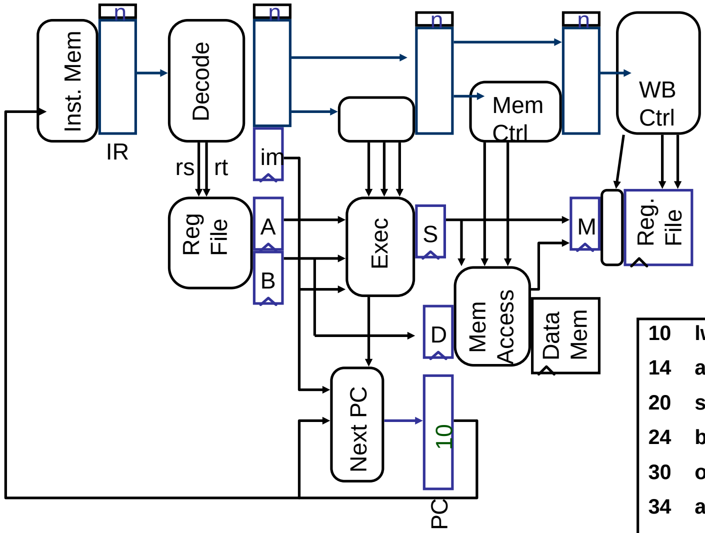


Let's Try it Out

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

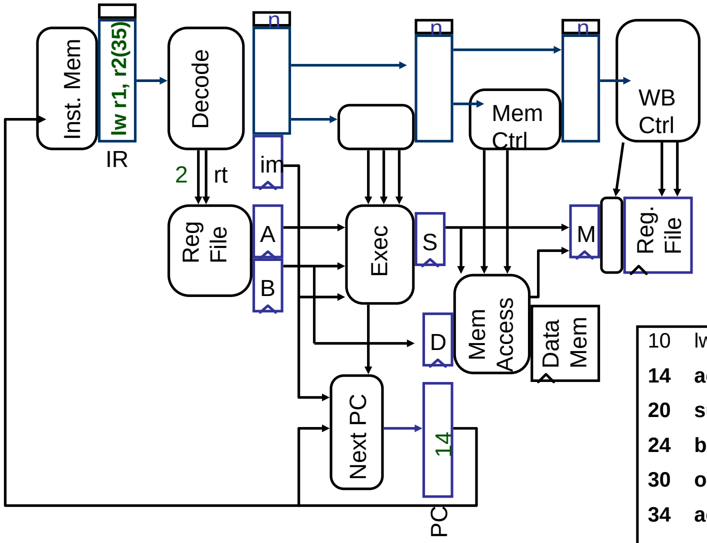
Address is in octal

Start: Fetch 10



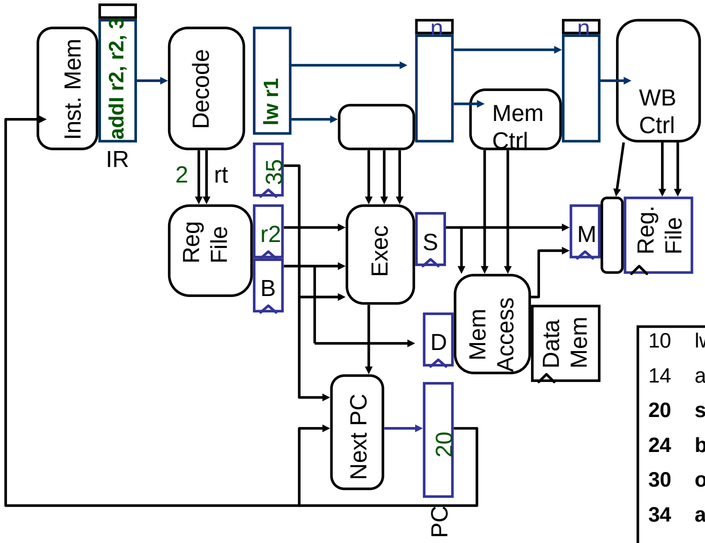
10	lw	r1, r2(35)
14	addI	r2, r2, 3
20	sub	r3, r4, r5
24	beq	r6, r7, 100
30	ori	r8, r9, 17
34	add	r10, r11,r12
100	and	r13, r14, 15

Fetch 14, Decode 10



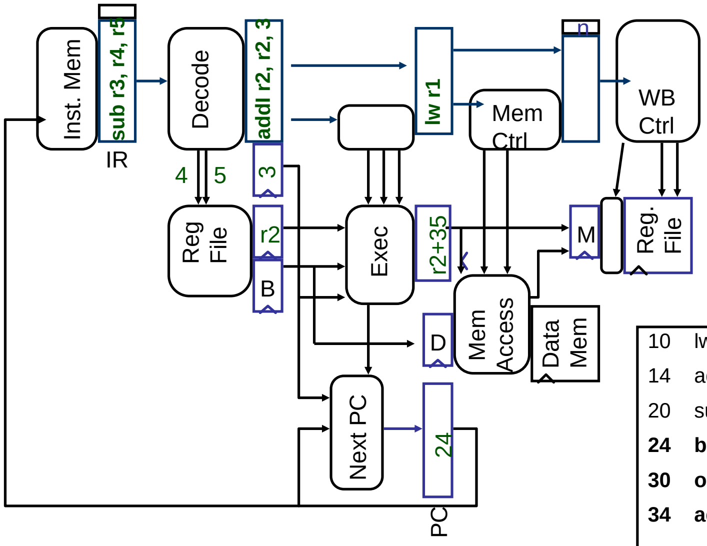
10	lw	r1, r2(35)
14	addI	r2, r2, 3
20	sub	r3, r4, r5
24	beq	r6, r7, 100
30	ori	r8, r9, 17
34	add	r10, r11,r12
100	and	r13, r14, 15

Fetch 20, Decode 14, Exec 10



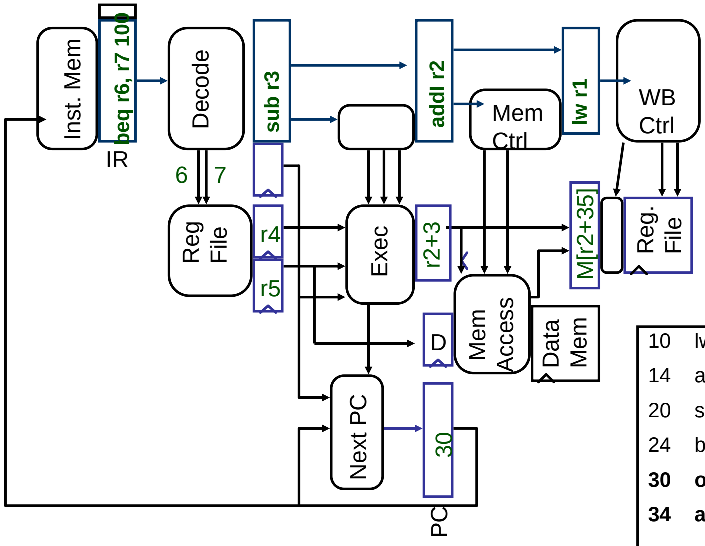
10	lw	r1, r2(35)
14	addI	r2, r2, 3
20	sub	r3, r4, r5
24	beq	r6, r7, 100
30	ori	r8, r9, 17
34	add	r10, r11,r12
100	and	r13, r14, 15

Fetch 24, Decode 20, Exec 14, Mem 10



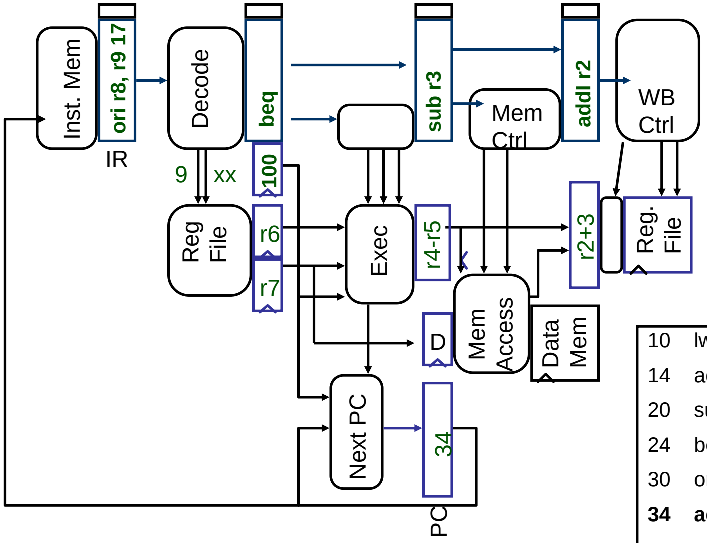
10	lw	r1, r2(35)
14	addI	r2, r2, 3
20	sub	r3, r4, r5
24	**beq**	**r6, r7, 100**
30	**ori**	**r8, r9, 17**
34	**add**	**r10, r11,r12**
100	**and**	**r13, r14, 15**

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**

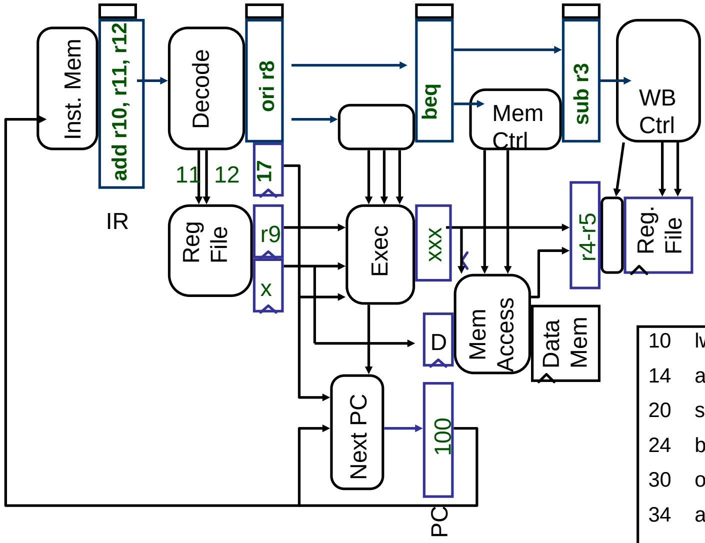
Fetch 34, Dcd 30, Ex 24, Mem 20, WB 14



r1=M[r2+35]

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 34, Ex 30, Mem 24, WB 20

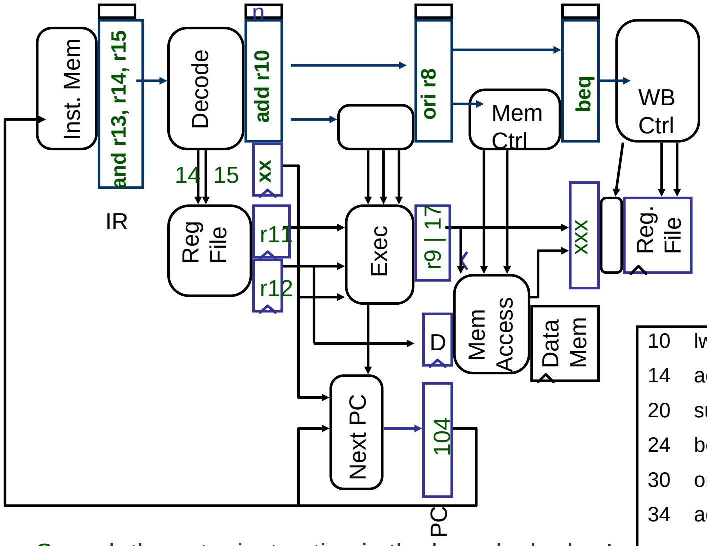


r1=M[r2+35]
r2 = r2+3

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

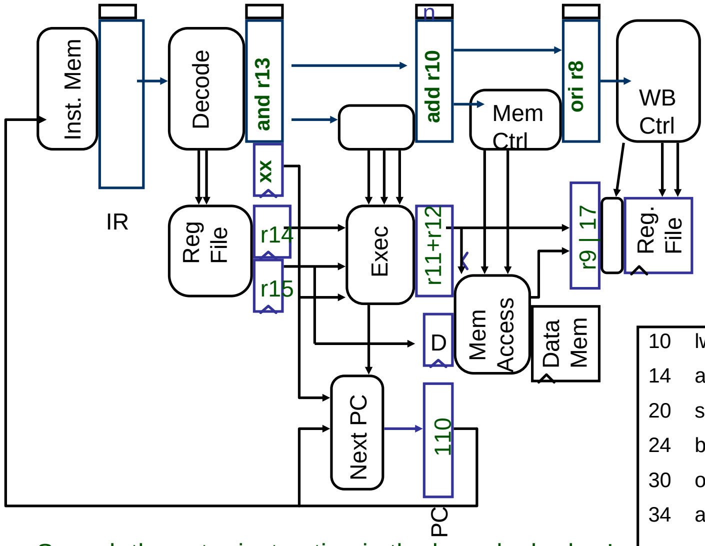


r1=M[r2+35]
r2 = r2+3
r3 = r4-r5

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

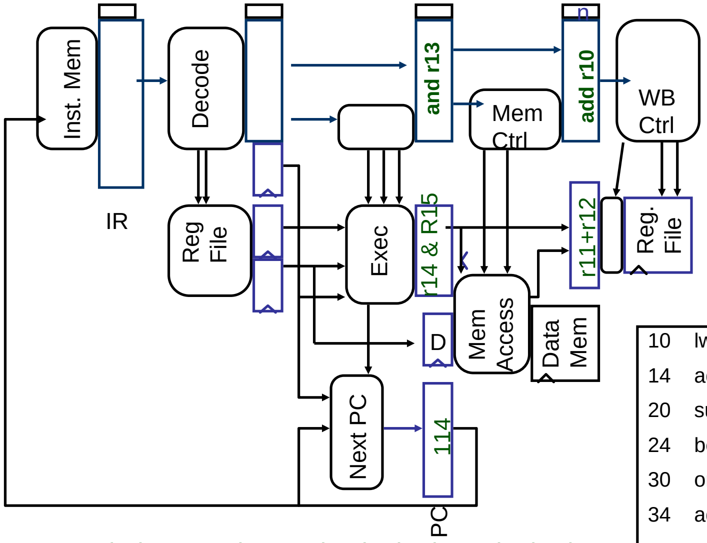


r1=M[r2+35]
r2 = r2+3
r3 = r4-r5

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



NO WB
NO Ovflow

r1=M[r2+35]
r2 = r2+3
r3 = r4-r5
r8 = r9 | 17

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!

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 (cont.)

- We'll build a simple pipeline and look at these issues
- We'll talk about modern processors and what really makes it hard:
  - exception handling
  - trying to improve performance with out-of-order execution, etc.

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