Review: (Write on the board before class)
1) Quick review of DLX implementation
2) CPI about 5 to 6 with straightforward implementation
5 steps:
1. Instruction fetch step:
   MAR ← PC; IR ← M[MAR]
2. Instruction decode/register fetch step:
   A ← Rs1; B ← Rs2; PC ← PC + 4
3. Execution/effective address step:

Memory reference:
   MAR ← A + $(IR_{16})^{16}$##$IR_{16..31}$; MDR ← B

ALU instruction:
   ALUoutput ← A *op* (B or $(IR_{16})^{16}$##$IR_{16..31}$)

Branch/Jump:
   ALUoutput← PC+$(IR_{16})^{16}$##$IR_{16..31}$; cond←(A *op* 0)
4. Memory access/branch completion step:

Memory reference:
   MDR ← M[MAR] or M[MAR] ← MDR;

Branch:
   if (cond) PC ← ALUoutput (branch)
5. Write-back step:
   Rd ← ALUoutput or MDR

Ch 6 Pipelining
DLX, interger only; later talk about floating point
assembly line, each in one piece; each called a *pipe stage* or *pipe segment*

Mainly Idea: little extra processor hardware 3X to 4X speedup

Throughput, not latency of a given instruction (time of instruction same)

Limited by slowest time; See Fig 6.2

Look at what happens each clock cycle;Make sure don't try to use hardware twice in one cycle or in wronf phase
Question: What are the resource conflicts?
1) PC incremented every clock => PC incrementer, not ALU
2) Must move up incrementing PC to first stage (otherwise on branch will not increment PC for next isntruction!)
3) New instruction fetched every clock (IF) and new data word fetched every clock on loads (MEM) => 2 ports
4) Separate path B to MDR for Store

5) Separate MAR for instructions and data (IMAR & DMAR)

Fig 6.3 & 6.4
6.3 what happens each pipe stage in the execution units ("reservation tables")
6.4 what happens each pipe stage per instruction; make sure doen't have conflicts between types of instructions in the pipeline at the same time

5X increase in memory system

Hazards => *stall* or *bubble* in pipeline Formula:

$$\text{Speedup} = \frac{\text{Clock cycle without pipelining}}{\text{Clock cycle with pipelining}} * \frac{\text{Ideal CPI} * \text{Pipeline depth}}{\text{Ideal CPI + Pipeline stall cycles}}$$

TYpically assume first term is 1

3 kinds of Hazards
Structural: resource conflicts: HW cannot support this combination of instructions
e.g., single memory port results in stalls for loads or stores
Data: dependency between instructions exposed by pipeline overlap

```
ADD   R1,R2,R3
SUB   R4,R1,R5

See Figure 6.6
```

Solution: Forwarding Figure 6.9 (see 6.8)
Classifications:
Read After Write; j reads before i writes
Compiler people call "dependency"
Write After Read: j writes before i reads
Compiler people call "antidependency"
Write After Write: i writes before i writes
Compiler people call "output dependency"
Hardware must make it work properly; compilers, programmers try to avoid case so it runs fast

Why no WAR & WAW in DLX?
All reads in ID stage (can't write first)
All writes in WB stage (can't write first)

Question: Would autoincrement would make it possible?
Yes: Take a look
WAW:

ADD R1,R2,R3
LD R4,R1+

Clear problem: EX phase of Load changes R1 before WB phase of ADD

WAR:
ADD R1,R2,R3
LD R4,R2+

Can't be a problem: you've only added another phase wher writes occur, reads still happen early

Can't Solve loads with forwarding; see 6.10
Solution: HW interlock to garuentee execution, SW to scheudle to avoid interlocks

Always in SW called delayed load
6.13 shows how effective

Control: pipeline branches (PC changing)

Smilar problem & solution

SUMMARY: Little change to processor (extra incrementor), big change to memory => 3 to 4 times speedup

BREAK: Project Survey/proposal due Friday;
handout signups today?
Learn 3 names
Next assignment? Ch 6 : 6.2, 6.3, 6.4

MONDAY February 26, 1990 1st Meeting Signup

Slot	Time	Name1	Name2	Proj. No.	Project Title
1	8:30 AM				
2	8:38 AM				
3	8:46 AM				
4	8:54 AM				
5	9:02 AM				
6	9:10 AM				
7	9:18 AM				
8	9:26 AM				
9	9:34 AM				
10	9:42 AM				
	BREAK				
11	10:10 AM				
12	10:18 AM				
13	10:26 AM				
14	10:34 AM				
15	10:42 AM				
16	10:50 AM				
17	10:58 AM				
18	11:06 AM				
19	11:14 AM				
20	11:22 AM				

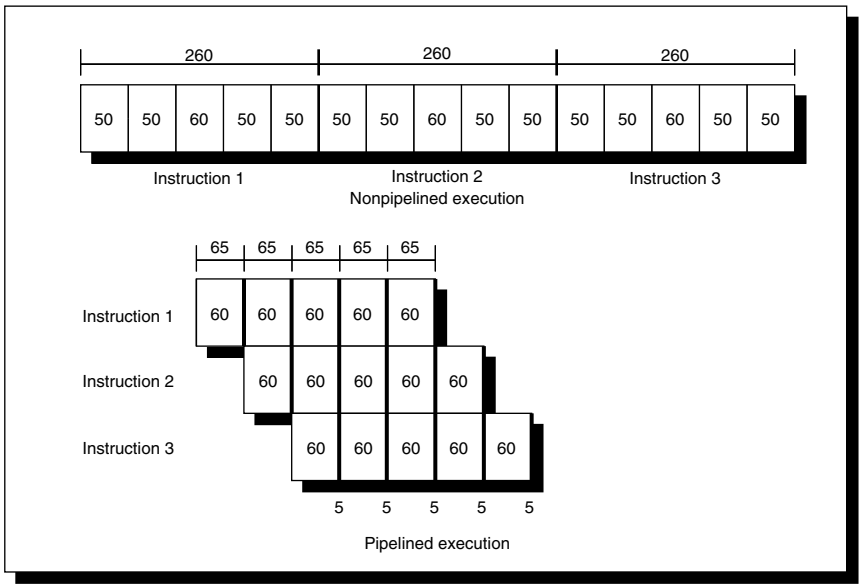

FIGURE 6.2
Note limited by slowest clock cycle + latch time
260 vs. 65 = 4X

Stage	PC unit	Memory	Data path
IF	IMAR← PC; PC←PC+4;	IR←Mem[IMAR];	
ID			A ← Rs1; B ← Rs2;
EX			DMAR← A + $(IR_{16})^{16}$##$IR_{16..31}$; or ALU_{output} ← A *op* (B or $(IR_{16})^{16}$##$IR_{16..31}$); or ALU_{output}←PC + $(IR_{16})^{16}$##$IR_{16..31}$; cond← (Rs1 *op* 0); MDR←B;
	if (cond) PC←ALUoutput	MDR← Mem[DMAR] or Mem[DMAR]←MDR	
WB			Rd ← ALUoutput or MDR

FIGURE 6.3 The table shows the major functional units and what may happen in every pipe stage in each unit.

Stage	ALU instruction	Load or store instruction	Branch instruction
IF	IMAR←PC; IR←Mem[IMAR]; PC←PC+4;	IMAR←PC; IR←Mem[IMAR]; PC←PC+4;	IMAR←PC; IR←Mem[IMAR]; PC←PC+4;
ID	A←Rs1; B←Rs2;	A←Rs1; B←Rs2;	A←Rs1; B←Rs2;
EX	ALUoutput ← A *op* B; or ALUoutput←A *op* $((IR_{16})^{16}$##$IR_{16..31})$;	DMAR← A+ $((IR_{16})^{16}$##$IR_{16..31})$; MDR← B;	ALUoutput←PC + $((IR_{16})^{16}$##$IR_{16..31})$; cond←(Rs1 *op* 0);
MEM		MDR← Mem[DMAR]; or Mem[DMAR]←MDR;	if (cond) PC←ALUoutput;
WB	Rd ← ALUoutput;	Rd ← MDR;	

FIGURE 6.4 Events on every pipe stage of the DLX pipeline

$$\text{Speedup} = \frac{\text{Clock cycle without pipelining}}{\text{Clock cycle with pipelining}} * \frac{\text{Ideal CPI} * \text{Pipeline depth}}{\text{Ideal CPI} + \text{Pipeline stall cycles}}$$

Instruction	Clock cycle 1	2	3	4	5	6
`ADD` instruction	IF	ID	EX	MEM	WB–data written here	
`SUB` instruction		IF	ID–data read here	EX	MEM	WB

FIGURE 6.6

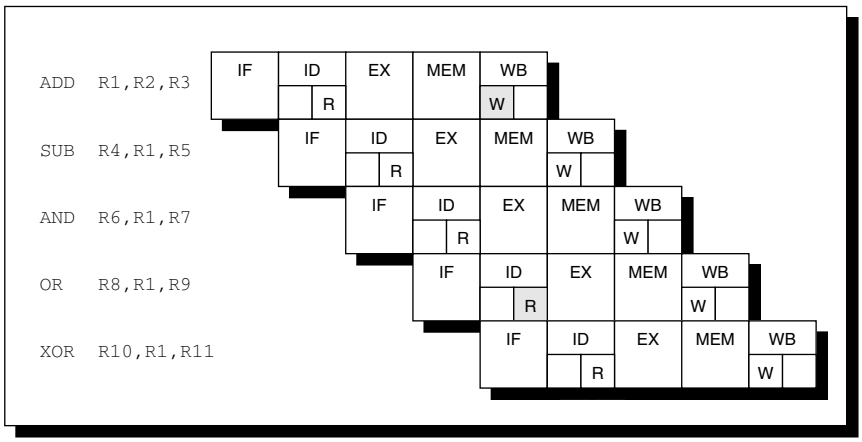

FIGURE 6.8

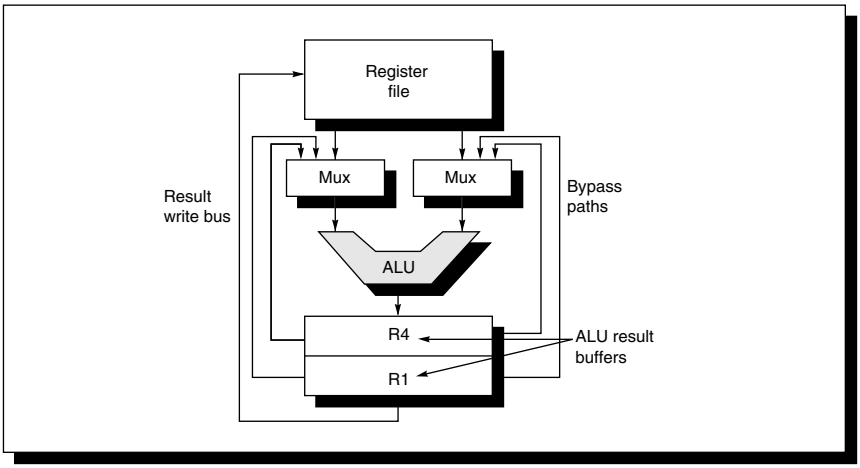

FIGURE 6.9

`LW R1,32(R6)`	IF	ID	EX	**MEM**	WB	
`ADD R4,R1,R7`		IF	ID	**EX**	MEM	
`SUB R5,R1,R8`			IF	ID	**EX**	
`AND R6,R1,R7`				IF	**ID**	

FIGURE 6.10

Any instruction	IF	ID	EX	MEM	WB					
`LW R1,32(R6)`		IF	ID	EX	**MEM**	WB				
`ADD R4,R1,R7`			IF	ID	*stall*	**EX**	MEM	WB		
`SUB R5,R1,R8`				IF	*stall*	**ID**	EX	MEM	WB	
`AND R6,R1,R7`					*stall*	IF	ID	EX	MEM	WB

FIGURE 6.11

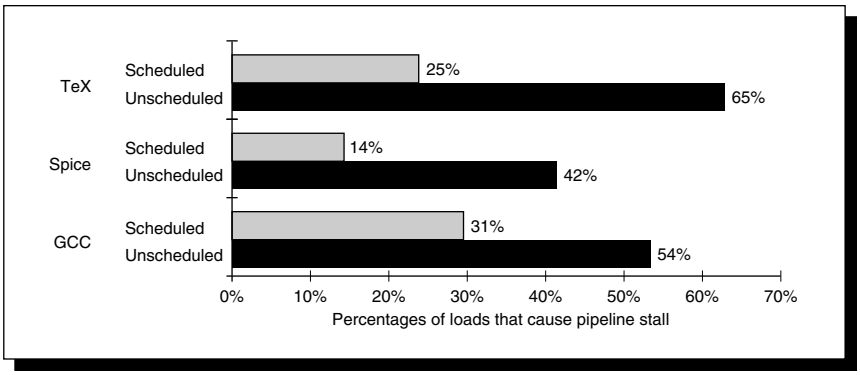

FIGURE 6.13 Percentage of the loads that result in a stall with the DLX pipeline.