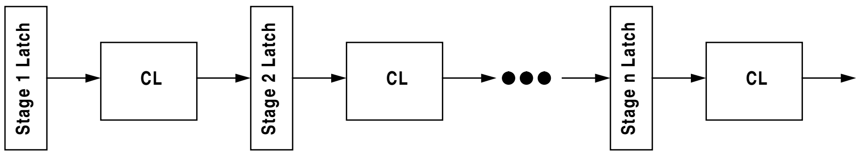
Pipelining

- ❑ Computational assembly line
  - each step does a small fraction of the job
  - all steps ideally operate concurrently
- ❑ A form of vertical concurrency
- ❑ Stage/segment - responsible for 1 step
  - 1 machine cycle per stage - usually equal to 1 clock cycle
  - synchronous design ==> slowest stage dominates
  - hence need for laminar stage design
- ❑ Simple model - common latch clock



Major Pipeline Benefit = Performance

- ❑ Ideal Performance
  - time-per-instruction = unpiped-instruction-time/#stages
  - asymptote of course - however +10% is commonly achieved
  - difference is due to difficulty in achieving laminar stage design

2 ways to view the performance mechanism

- ❑ Reduced CPI (valid in non-piped to piped architecture change)
  - sequence changes due to overlapped execution constraints
  - average is reduced
- ❑ Reduced cycle-time (valid when increasing the pipeline depth
  - work split up into multiple stages
  - simpler stages result in faster clock cycles

Other Pipeline Benefits

- ❑ Completely HW mechanism
- ❑ No programming model shift required to exploit this form of concurrency
  - except for performance programmer is oblivious to the organization change
  - BUT - the compiler will need to change and it compile time will go up
- ❑ All modern machines are pipelined
  - key technique in advancing performance in the 80's
  - in the 90's we just moved to multiple pipelines
- ❑ Beware - no benefit is totally free/good

Start with Unpipelined DLX

Every DLX instruction can be XEQ'd in 5 steps

- ❑ Unpipelined case:
  - all 5 steps will occur in the same clock
  - each step outputs just passed to next step (no latches)
- ❑ Remember the instruction set

0 5	6 10	11 15	16 31
Opcode	rs1	rd	immediate

I-type: loads and stores, R-immed ALU ops, Condix branch (rs1 and rd unused, Jump reg, J&Link reg

0 5	6 10	11 15	16 20	21 31
Opcode	rs1	rs2	rd	function

R-type: reg-reg ALU ops (func=op) read/write special regs, and moves

Opcode	Offset added to PC

J-type: Jump and Link, Jump, TRAP, and RFE

Steps 1 & 2

- ❑ IF - instruction fetch step
  - IR <-- Mem[PC] --------- fetch the next instruction from memory
  - NPC <-- PC + 4 ---------- compute the new PC
  - done in parallel with opcode decode
- ❑ ID - instruction decode and register fetch step
  - A <-- Regs[$IR_{6..10}$]
  - B <-- Regs[$IR_{11..16}$]
  - Possible since register specifiers are encoded in *fixed fields*
  - We may fetch register contents that we don't use but OK since the operands will be ready if the opcode is of the type that does use them
  - Also calculate the sign extended immediate in case that's the value that the opcode needs

Step 3

EX - execution/effective address step

- ❑ There are 4 options depending on the opcode decoded in the previous step
  - Memory Reference
    - ALUOutput <-- A + $(IR_{16})^{16}$ ## $IR_{16..31}$ --------- effective address
    - SMD <-- B --------- data to be written if it is a STORE -- SMD (store mem data) = MDR
  - Register - Register ALU instruction
    - ALUOutput <-- A op B
  - Register - Immediate ALU instruction
    - ALUOutput <-- A op (($IR_{16})^{16}$ ## $IR_{16..31}$))
  - Branch/Jump
    - ALUOutput <-- NPC + $(IR_{16})^{16}$ ## $IR_{16..31}$
    - actually a 16 bit immed. for branch - 26 bit immed for a jump
    - cond <-- A op 0 --- for conditional branches A's value is the condition base (= for BEQZ)

In simple Load-Store machine no instruction needs to calculate both a memory address and an instruction address and do an ALU op- hence EX/EFA can be combined into a single cycle.

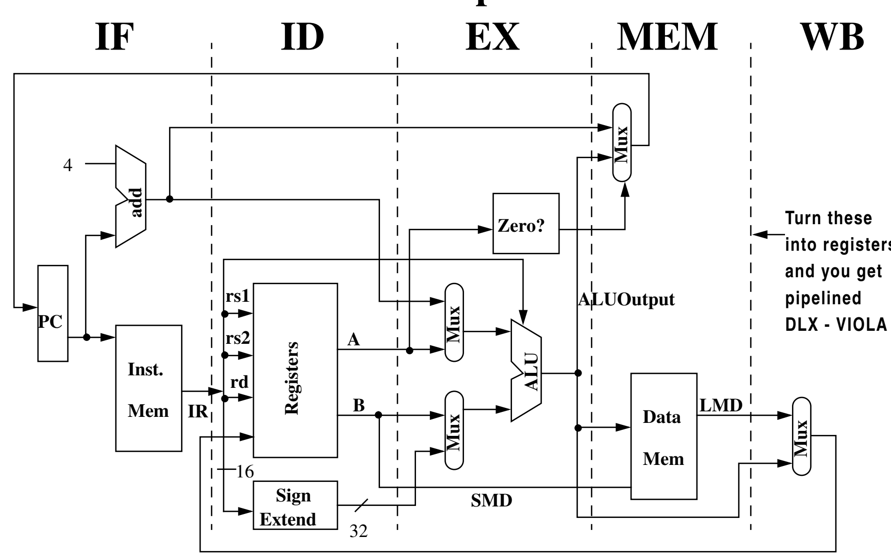
Steps 4 & 5

- ❑ MEM - memory access/branch completion
  - Memory Reference
    - LMD <--- Mem[ALUOutput] ------- if it's a load; LMD (load memory data) = MDR

OR

    - Mem[ALUOutput] <-- SMD
  - Branch
    - if (cond) then PC <-- ALUOutput else PC <-- NPC
    - for Jumps the condition is always true
- ❑ WB - write back
  - Reg - Reg ALU
    - Regs[$IR_{16..20}$] <-- ALUOutput
  - Reg - Immed ALU
    - Regs[$IR_{11..15}$] <-- ALUOutput
  - Load
    - Regs[$IR_{11..15}$] <-- LMD

DLX Datapath



How Real is This?

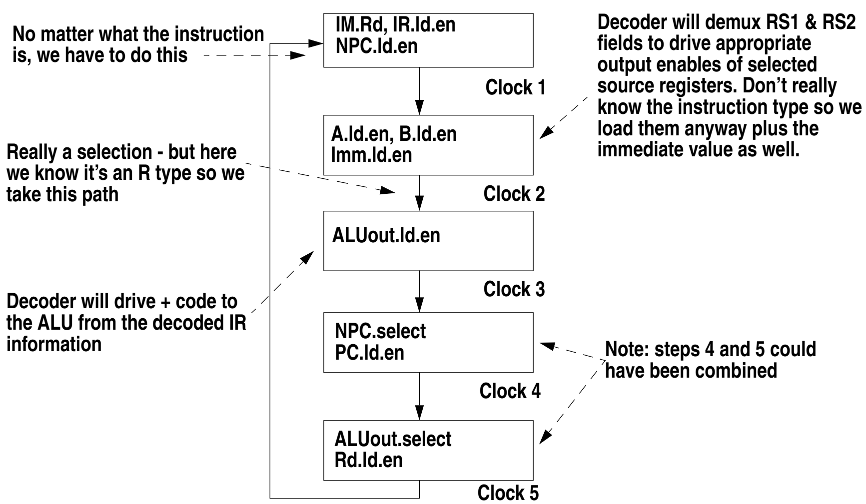
- Close
  - each stage will have a register separating it from the next stage
- Model is correct but not optimized
  - OK since this is the next step
- There are other choices
  - 2 ALU’s - 1 would have sufficed since in any given cycle only 1 is active
  - instruction and data memories do not have to be separate
  - could have gone for 1 long cycle rather than 5 short ones
- Biggest Flaw - no control
  - finite state machine or microcode
    - microcode is extinct and likely to remain so

What Does Control Do?

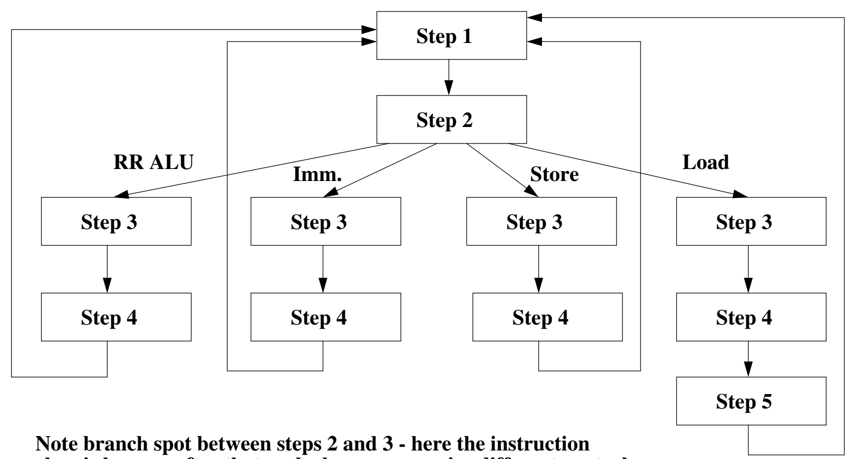
- Clock pulse says when but control must say OK and in some cases what
  - registers have load-enable control lines
  - memories have R/W and output-enable lines
  - mux’s provide date steering but they have select lines
    - check out COND which provides steering for proper new PC value
    - note that the write-back MUX stage has no selector (this comes from the control)
- Control issues are mostly ignored in the book
  - details viewed as an implementation issue
  - they are however complex and important
  - hence we’ll treat the details at a higher level of abstraction
- Microcode is dead - any ideas as to why?
  - book seems to want to view it as a viable option

Sample Control (Moore FSM style)

Rx <-- Rx + Ry



Full Control Scenario



Note branch spot between steps 2 and 3 - here the instruction class is known, after that each class may require different control point assertions. Also note that only the load requires all 5 cycles.

DLX Pipeline Example

- Stages now get executed 1 per cycle
  - rather than 5 cycles for the whole thing and then repeat
  - ideal result is the CPI went from 5 to 1
  - Is it really this simple?
    - of course not but it's a start
    - e.g. what happens if you want to use the same unified instruction and data memory?
    - things will grow more hair soon - e.g. exceptions, bypass, forwarding, etc.
- Best case pipeline scenario

Instruction					Clock	Number			
Number	1	2	3	4	5	6	7	8	9
i	IF	ID	EX	MEM	WB				
i+1		IF	ID	EX	MEM	WB			
i+2			IF	ID	EX	MEM	WB		
I+3				IF	ID	EX	MEM	WB	
i+4					IF	ID	EX	MEM	WB

Important Pipeline Characteristics

- Latency
  - time it takes for an instruction to go through the pipe
  - latency = # stages x stage-delay
  - dominant feature if there are lots of exceptions
- Throughput
  - determined by the rate at which instructions can start/finish
  - dominant feature if no exceptions

Benefit Example

- Unpipelined DLX
  - 5 steps - take 50, 50, 60, 50, 50 ns respectively
  - hence total instruction time = 260 ns
- Looks like a 5-stage pipeline
- But there are parasites everywhere
  - assume 5 ns added to slowest stage for extra latches
  - primarily due to set-up and hold times
- Hence (assuming no stage/step improvement)
  - must run at slowest stage + parasitics = 60 + 5 = 65 ns/stage
  - in steady state (no exceptions) instruction done every 65 ns
- Speedup 260/65 = 4x improvement

All that glitters ....

- Pipelines introduce some real hair of course
  - making every stage laminar is hard
  - concurrency ==> no shared resources between stages
    - to start the analysis
    - list resource path of each stage
    - look for conflicts - remember arbitrary instructions may be in the pipe at the same time
    - duplicate resources so each stage has it private set
  - for example
    - PC must be incremented every cycle and can't use the same ALU as arith. ops
    - new instruction must be fetched - can't compete with data memory accesses of other stages: i.e. not conflict in IF and MEM stages
    - loads and stores overlap needs for MDR - hence need two - LMDR and SMDR
    - latches which hold next-PC, instruction, and ALU output are used later but may get hammered by subsequent instruction
  - What about memory bandwidth
    - memory latency hasn't changed but bandwidth demand is up to 5x for 5 stage pipe
    - Harvard architecture helps - separate I & D memories
    - multi-level caches - more on this later

... is not Gold

- Life is pretty good when pipe members are independent
- Sadly this is seldom true
- e.g. R5 <-- R2 + R3, R3<-- R5 + R6
  - write back of R5 doesn't happen until stage 5 for inst. i
  - too bad it's needed in stage 2 of inst i + 1
  - OOPS - problem - called pipeline hazard
- Enter bypass and stalls
  - stages must communicate (bypass) or stall
  - bypasses mean more logic + more control
  - stall dependencies mean the same
- Imagine the effect on the data and control paths

Pipeline Hazards

- Structural hazards
  - caused by resource contention
    - single port register file - conflict with multiple stage needs
    - memory fetch - may need one in both IF and MEM stages
  - possible to avoid by adding resources
  - may be too costly
- Data hazards
  - instruction needs unavailable result from previous stage
  - can be mitigated somewhat by a smart compiler
- Control hazards
  - when the PC doesn't get just incremented
  - branches and jumps - not too bad
  - interrupts - Aaaaarrrrrrrggggggghhhhhhhhh!

Simple DLX Pipeline Activities

from the major functional unit perspective

Stage	PC Unit	Memory	Data Path
IF	PC <-- PC + 4	IR <-- Mem[PC]	
ID	PC1 <-- PC	IR1 <--IR	A <-- Rs1; B<--Rs2
EX			DMAR <--A + $(IR1_{16})^{16}$ ## $IR1_{16..31}$ or ALUout <-- A op B or ALUout <-- A op $(IR1_{16})^{16}$ ## $IR1_{16..31}$ or ALUout <-- PC1 + $(IR1_{16})^{16}$ ## $IR1_{16..31}$ or cond <-- (Rs1 op 0); SMDR <-- B
MEM	if (cond) then PC <-- ALUout	LMDR <-- MEM[DMAR] or MEM[DMAR] <-- SMDR	ALUout1 <-- ALUout
WB			Rd <-- ALUout1 or Rd <-- LMDR

Simple DLX Pipeline Activities

from an instruction class perspective

Stage	ALU instruction	Load or Store	Branch
IF	IR <-- MEM[PC]; PC <-- PC+4	IR <-- MEM[PC]; PC <-- PC+4	IR <-- MEM[PC]; PC <-- PC+4
ID	A<--Rs1; B<--Rs2; PC1<--PC; IR1<--IR	A<--Rs1; B<--Rs2; PC1<--PC; IR1<--IR	A<--Rs1; B<--Rs2; PC1<--PC; IR1<--IR
EX	ALUout <-- A op B or ALUout<--A op $(IR1_{16})^{16}$ ## $IR1_{16..31}$	DMAR<--A+$(IR1_{16})^{16}$ ## $IR1_{16..31}$: SMDR <-- B {if it's a store}	ALUout <-- PC1+$(IR1_{16})^{16}$ ## $IR1_{16..31}$ cond <-- Rs1 op 0
MEM	ALUout1 <-- ALUout	LMDR <-- MEM[DMAR] or MEM[DMAR] <-- SMDR	if (cond) then PC<--ALUout
WB	Rd <-- ALUout1	RD <-- LMDR {if it's a load}	

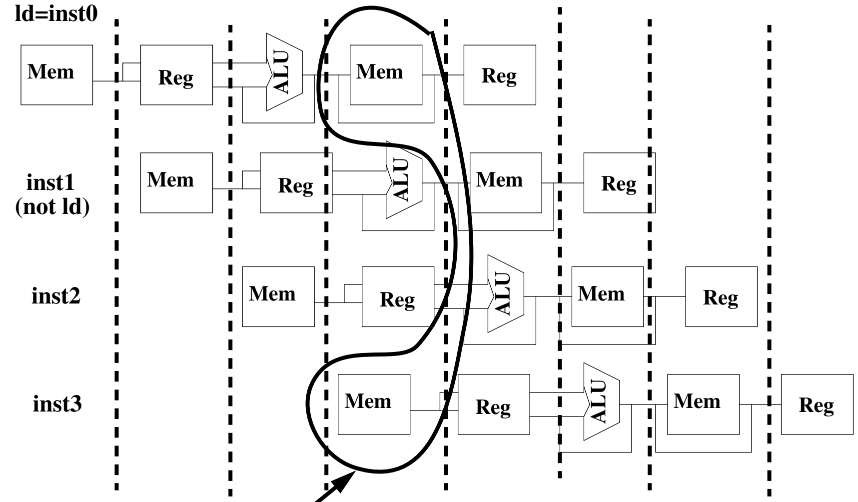
- note potential stage holes where nothing much happens
- note pre-decode (stages 1 & 2) - same for all

Hazards cause Stalls

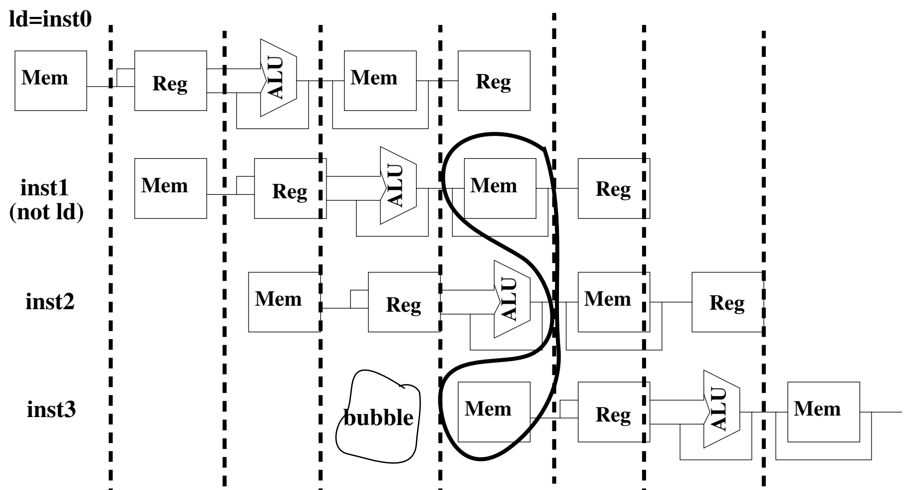
Two Policy Choices

- How about just stalling all stages
  - OK but problem is usually adjacent stage conflicts
  - hence nothing moves and stall condition never clears
  - cheap option but it doesn't work
- Stall early let late progress
  - stages preceding stalled stage can't go anywhere
    - once the problem clears the stall can be cleared
    - this is the role of stall-control blocks that control each stage
  - later stages allow to complete
  - works even better in dynamic pipeline configurations
    - hairier still
    - but not every stage is always necessary - hence stage skipping is a possibility
    - controlling the possibility is the problem since control complexity goes up

Pipe from a Resource View



Stalls Create Bubble - Removes Hazard



no hazard if inst1 is not a load

Calculating Stall Effects

$$\text{Pipeline Speedup} = \frac{\text{Average instruction time without pipelining}}{\text{Average instruction time with pipelining}}$$

$$\text{Pipeline Speedup} = \frac{\text{unpiped cycle time}}{\text{piped cycle time}} \times \frac{\text{unpiped CPI}}{\text{piped CPI}}$$

$$\text{Ideal CPI} = \frac{\text{unpiped CPI}}{\text{Pipeline Depth}}$$

Therefore

$$\text{Pipeline Speedup} = \frac{\text{unpiped cycle time}}{\text{piped cycle time}} \times \frac{\text{Ideal CPI} \times \text{Pipeline Depth}}{\text{piped CPI}}$$

Calculating Further

However

$$\text{piped CPI} = \text{Ideal CPI} \times \text{Pipeline stall cycles} = 1 + \text{average stalls per instruction}$$

Then if perfect balance: no overhead and cycle times equal

$$\text{Speedup} = \frac{\text{CPI Unpiped}}{1 + \text{Pipeline stall cycles per instruction}}$$

If laminar then unpiped CPI = pipeline depth, hence

$$\text{Speedup} = \frac{\text{Pipeline Depth}}{1 + \text{Pipeline stalls per instruction}}$$

Similar derivations for clock cycles are also possible

However

the no overhead model is basically bogus

- ❑ Bew are c hanges in cyc le time
  - latc h overheads are real
- ❑ Bew are of stall effects
  - the deeper the pipe the more the replicated resources OR
  - the broader the stall c hance
- ❑ Amdahl' s la w type effect
  - no guessing - you ha ve to run the simulations

The plot sickens from here