Lecture #5

****
Review -- 1 min
****

Pipelining -- as natural as doing laundry
Simple instructions, Simple Datapath
Limits to pipelining

- helps throughput
- limited by slowest stage
- parallelism
- max speedup = # pipe stages
- "fill" and "drain" time limit speedup

****
Outline - 1 min
****

<good news/bad news>

- Speedup Equation
- 3 Hazards
  - structural
  - data
  - control
- Data Hazards

****
Lecture - 20 min
****

Basic Pipelining

It's natural!

Laundry example

- I have 4 loads
- Washer: 30 minutes

Dryer: 40 minutes
Folding: 20 minutes

Sequential -- unpipelined

Load1: Wash Dry Fold
<---30--> <----- 40 -----> <---20--->
Load2: Wash Dry Fold
<--30--> <-----40-----> <--20-->

etc <draw rest on board>....

--> Total time for 4 loads -- 6 hours

Would never do it like that!!

Pipelined
Load1: Wash Dry Fold
<--30--><----40----> <--20->
Load2: Wash Dry Fold
<--30--><-10-> <----40----> <--20-->
Load3: Wash Dry Fold
<--30--><-10-> <----40----> <--20-->
Load4: Wash Dry Fold
<--30--><-10-> <----40----> <--
20-->

Total time: 30 + 40 + 40 + 40 +40 +20 = 210 = 3.5 hours

Laundry Lessons

1. Doesn't help latency of single task (helps throughput)
2. rate limited by slowest stage
3. multiple tasks operating in parallel
4. potential speedup = # pipe stages
5. unbalanced lenghts of stages reduce speedup
   (b/c rate limited by slowest)
6. time to "fill" and "drain" pipeline reduces speedup

<picture of DLX data path>

MIPS Instruction Set

32-bit fixed instruction format
32 registers (r0 contains 0)
3 address reg-reg ALU instructions
load/store
simple branch conditions

Register-Register
    Op Rs1 Rs2 Rd Opx--
Register-Immediate
    Op Rs1 Rd immediate------
Branch
    Op Rs1 Rs2/Opx immediate------
Jump/Call
    Op Target---------------------------------

<step through how each type of instruction works
using picture of data path and instruction types>

Unpipelined -- <CLOCK>

start an instruction
wait for logic to work itself out
<CLOCK> latch result in register
start next instruction

<picture of DLX datapath with pipeline registers>

<CLOCK> start I1
<CLOCK> start I2 ID I1
<CLOCK> ...

****
Admin - 3 min
****

discuss project interests
anyone need project partner?

HW1 due today. 5pm

LESSTR

****
Lecture - 24 min
****

Speed Up Equation for Pipelining

Speedup = Time_unpipelined / Time_pipelined

CPU_Time = #Instr * CPI * Clock Cycle

#INstructions unchanged

$$\text{Speedup} = \frac{\text{IdealCPI * Pipeline Depth}}{\text{IdealCPI + Pipe stall CPI}} * \frac{\text{ClockCycle\_unpipelined}}{\text{Clock Cycle\_pipelined}}$$

-----> <show picture of pipelined datapath to explain equation>

Why "IdealCPI * Pipeline Depth"?
Why would clock cycle change?
Why are there "pipe stall CPI"?

For this chapter,

IdealCPI = 1

$$\text{Speedup} = \frac{\text{PipelineDepth}}{\text{1+PipeStallCPI}} * \frac{\text{ClockCycle\_unpipelined}}{\text{ClockCycle\_pipelined}}$$

Why not have a pipeline that is 1000 deep?

Limits to pipelining: Hazards cause Pipeline Stalls
--> Increase PipeStallCPI

3 types of hazard

1. Structural Hazard -- no hardware available
   need memory interface that can accept 1000 requests
   as quickly as it now accepts 5

2. Data hazard -- instruction depends on result of prior instruction
   Laundry lesson -- speedup comes from parallelism

to have 1000-deep pipeline, need 1000 instructions that you can execute in parallel
--> stall pipeline (e.g. stop) if instruction depends on any of the 1000 preceding instructions
(if you divide laundry into 1000 loads, how will you fold the socks?)

3. Control Hazard -- pipelining of instructions that change PC (e.g. branch)
Laundry lesson -- <the fill/drain laundry problem>

drain/fill pipeline (1000 cycles) on every branch

Relative Impact of Hazards
Structural -- not all that interesting
--> just spend more $'s on HW
(Tech trend -- # transistors going up fast)

Data hazard --

problem: dependencies among instructions

a = b + c
d = a + g

Can't calculate d until you know a

These dependencies put limit on how quickly you could execute a program
no matter how much hardware you have

Solution: execute second equation as soon as possible after first

Control hazard -- branches

hard problem -- basic blocks in computer programs are short

Solution -- start filling pipeline again as soon after a branch as possible
(avoid complete fill/drain)

Talk about the 3 types of hazard

Structural Hazards

Book steps through example -- suppose you can only do one memory access per cycle
--> MEM phase of a load stalls FETCH the next instructions
--> solution = separate I and D cache

Not very interesting -- just throw more HW at the problem

Architectural question -- is extra hardware worth it?

General approach
1. calculate speedup with and without hazard
2. calculate cost of eliminating hazard
3. compare cost/performance

--> all current processors have separate I and D caches
--> most current processors do not fully pipeline floating point units

Data Hazards
----

add r1, r2, r3	IF	ID	EX	MEM	WB			
sub r4, r1, r5		IF	ID	EX	MEM	WB		
and r6, r1, r7			IF	ID	EX	MEM	WB	
or r8, r1, r9			IF	ID	EX	MEM	WB	
xor r10, r1, r11				IF	ID	EX	MEM	WB

Straw man solution -- stall

add r1, r2, r3	IF	ID	EX	MEM	WB					
sub r4, r1, r5		stall	stall	stall	stall	IF	ID	EX	MEM	WB
and r6, r9, r7							IF	ID	EX	

...

Better solution: forward

<draw arrows in picture 1 -- add instruction knows answer after EX stage
--> "forward" result to later instructions>

How does forwarding work?

<use datapath + pipeline registers picture from last lecture>

1. result of EX stored in latch
2. add path to forward that result to input of EX
3. create paths from EX/MEM --> EX, MEM/WB-->EX,
   also, if WB in first part of cycle, can read registers in second part

<optional: depending on time>

Really, how does it work

if dest of prev instruction = one of my source registers, forward from EX/MEM to EX

if dest of 2-prev instruction = one of my source registers, forward from MEM/WB to EX

<end optional>

Load Stalls -- data hazard even with forwarding
----

lw r1, 0(r2)	IF	ID	EX	MEM	WB				
sub r4, r1, r5			IF	ID	stall	EX	MEM	WB	
<next instruction can't proceed b/c sub is stuck in pipeline>									
and r6, r7, r8				IF	stall	ID	EX	MEM	WB

Solution: Sofware scheduling
(don't use result of load in next instruction)
e.g.
lw r1, O(r2)
and r6, r7, r8
sub r4, r1, r5

Taxonomy of Data Hazards

<convention -- hazards named after the order that you must preserve
for correct operation>

RAW -- read after write
previous examples -- need a previous result
true dependency

WAR -- write after read

e.g. add r1, r2, r3
sub r2, r4, r5

--> must make sure that add reads r2 before sub overwrites it

Not a problem in DLX pipeline -- earlier instruction alwas
reads inputs before a later instruction writes outputs

antidependence

WAW -- write after write

e.g. add r1, r2, r3
sub r1, r4, r5

r1 had better end up with r4 - r5

output dependence

not a problem on basic dlx pipeline

Will see WAR and WAW later

Note: never see RAR

****

Summay - 1 min

****

<good news/bad news>

- Speedup Equation
- 3 Hazards
  - structural
  - data
  - control
- Data Hazards