CS 3220
Basic Pipeline
Why not go directly to five stages?
  – This is what we had in CS 2200!

Will have more stages in Project 3, but
  – We want to start with something easier
    • Lots of things become more complicated with more stages
    • Let’s first deal with simple versions of some of these complications
  – Will learn how to decide when/how to add stages
    • Start with two, then decide if we want more and where to split
• What gets done in which stage
  – Memory address for data reads must come from FFs
    • Memory read must be at the start of some stage
    • With only two stages, this has to be stage two!
  – Must be in first stage
    • Fetch, and all the other stuff needed for memory address:
      decode, read regs, ALU (or at least the add for memaddr)
  – Must be in last stage
    • Read memory, write result to register
  – Where does branch/jump stuff go
    • As early as possible (will see why) => first stage
Creating a 2-stage pipeline

Control

Instr Mem

RF

Data Mem

SE
assign dmemaddr = aluout;
reg [31:0] aluout_M;
always @(posedge clk)
  aluout_M <= aluout;
assign dmemaddr = aluout_M;
Two-Stage Pipeline

• So far we have
  – Stage 1: Fetch, ReadReg, ALU
  – Stage 2: Read/Write Memory, WriteReg

• What is left to decide?
  – Where is the PC incremented?
    • Input: PC (available at start of stage 1)
    • Work: Increment (doable in one cycle)
    • Do it in stage 1!

  – Where do we make branch taken/not-taken decisions?
    • Depends… try in cycle 1, but if this is critical path, try to break it up
• Our goal is to get this working!
  – Handling each type of hazard will complicate things
  – Avoid doing things that create hazards
• Structural hazards?
  – Noooo! We will put enough hardware to not have any!
• Control hazards?
ADD R1,R2,R3
ADD R4,R2,R1
• What happens in our two stage pipeline?
C1: aluout_M<=R2+R3
C2: R1<=aluout_M; aluout_M<=R2+R1 (problem!)
C3: R4<=aluout_M
LW  R1,0(R2)
ADD  R3,R1,R4

• What happens in our two stage pipeline?
C1: aluout_M<=0+R2
C2: R1<=mem[aluout_M]; aluout_M<=R1+R4
Preventing data hazards

• Simplest solution for HW designers
  – Tell programmers not to create data hazards!

ADD R1,R2,R3

<Instruction that does not use R1>
ADD R4,R2,R1

What if we have nothing to put here?

NOP
What is a NOP?

- Does not do anything
- How about AND R0,R0,R0?
  - Whatever is in R0, leaves it unchanged
- Why is this not a good NOP?
; Initially R0 (Zero) is some random value
XOR Zero,Zero,Zero
NOP ; Becomes AND R0,R0,R0
ADDI SP,Zero,StackTop  ➙ What is in SP now?
ADDI A1,Zero,1  ➙ What is in A1 now?
• **Actually** does nothing
  – Not just “writes the same value”
  – wrreg, wrmem, isbranch, isjump, etc. must all be zero!
• None of our instructions is a truly perfect NOP
• So let’s add one!
  – Hijack existing instruction, e.g. AND R0,R0,R0 ?
    • It works! This instruction is not supposed to do anything anyway!
  – Add a separate instruction (and spend an opcode)
    • Also works! But spend a secondary opcode
• Let’s use ALUR with op2=0 (and all other bits 0)
  – NOP translates to instruction word 32’h00000000
Control hazards

• No problem if all insts update PC in first stage
  – PC+4 is easy, but branches and jumps not so easy
  – What if PC+4 in cycle 1, but the rest in cycle 2
JAL RA,Func(Zero)  C1: PC<=PC+4  C2: PC<=Func
BNE RV,Zero,BadResult  C2: Fetch  C3: PC=<BadResult
ADD T0,RV,RV

...  
BadResult:  C4: Fetch

...  
Func:  C3: Fetch
Preventing control hazards

- Simplest solution for HW designers
  - Tell programmers that branch/jump has delayed effect
  - Delay slot: inst after branch/jump executed anyway

JAL RA,Func(Zero)
NOP ; Delay slot
BNE RV,Zero,BadResult
NOP ; Delay slot
...

CS 3220 Fall 2011 - Prof. Milos Prvulovic
Deeper pipelines

• Need more NOPs
  – More instructions between reg write and reg read
    • Hard to find useful insts to put there => NOPs
  – More delay slots to survive control hazards
    • Hard to find useful insts to put there => NOPs

• Problem 1: Performance
  – Note that CPI is 1, but program has more instructions!

• Problem 2: Portability
  – Program must change if we change the pipeline
    • What works for 2-stage needs more NOPs to run on 3-stage, etc.
Architecture vs. Microarchitecture

• Architecture
  – What the programmer *must* know about our machine

• Microarchitecture
  – How we implement our processor
  – Can write correct code without knowing this

• Our hazards solution
  – Pipelining = microarchitecture
  – Delay slots, etc. = architecture
  – We changed architecture (in a backward-incompatible way) to make our microarchitecture work correctly!
Proper handling of hazards

- Programs (executables) don’t change
  - Test2.mif, Sorter2.mif from Project2 still run correctly
- Must fight hazard problems in hardware
  - Our big weapon: “flush” an instruction from pipeline
  - Our better weapon: “stall” some stages in the pipeline
  - Our precision weapon: forwarding
    - Can’t fix everything, but helps reduce the number of flushed insts
What is a flush

• Flush an inst from some stage of the pipeline == convert the instruction into a real NOP

• Note: cannot flush any inst from any stage
  – Can’t flush inst that already modified architected state
    • E.g. if SW already wrote to memory, can’t flush it correctly
    • E.g. if BEQ/BNE/JAL already modified the PC, can’t flush it correctly

• To prevent hazards from doing damage
  – Must detect which instructions should be flushed
  – And then flush these instructions early enough!
The Rules of Flushing

• When we **must** flush an instruction
  – When not doing so will produce wrong result

• When we **can** flush an instruction
  – Almost any time we want (if early enough), but must guarantee *forward progress*
    • E.g. can’t just flush every single instruction as soon as fetched

• Lots of room between the can and the must
  – For performance, get as close to “must” as possible
  – For simplicity, may do some “can but not must” flushes
• Find out K, the worst-case number of NOPs
  – # of NOPs between insts that prevents all hazards
  – E.g. in out 2-stage pipeline it’s 1 NOP

• If stages numbered 1..N, we flush the first K stages whenever a non-NOP inst in stage K+1
  – E.g. in our 2-stage pipeline, we would flush stage 1 whenever a non-NOP is in stage 2
  – What is the resulting CPI for the 2-stage pipeline?
• Data hazards - when we don’t have to flush
  – If without flushing NOPs would not be needed
    • If inst in stage K+1 has wrreg=0,
      E.g. SW doesn’t need NOPs after it
    • If inst in stage K+1 writes to regno we don’t read
      E.g. ADD R1,R2,R3 can be safely followed by ADD R2,R3,R4
  – If forwarding or stalling fixes the problem
    • We’ll talk about this later

• Control hazards – when we don’t have to flush
  – If we fetched from the correct place,
    e.g. if we fetched from PC+4 and BEQ not taken
– For a pipeline FF between some stages A and M:
always @(posedge clk or negedge reset)
    if(reset)
        wrreg_M<=1'b0;
    else
        wrreg_M<=wrreg_A;

    flush_A?1'b0:wrreg_A;
Stalling

• Stops instructions in early stages of the pipeline to let farther-along instructions produce results
  – Creates a “bubble” (a NOP) between the stopped instructions and the ones that continue to move

• For data hazards, stalls can entirely eliminate flushes
  – The bubble NOP is like a NOP we inserted into the program
    • But without changing the program
  – Why is a stall better than a flush?
    • When flushing some stage S (because of a dependence), must also flush stages 1..S-1 (can’t execute insts out-of-order)
      – Adds S new NOPs to the execution
    • When stalling stage S, must also stall stages 1..S-1
      – But each stall cycle inserts only one NOP (in stage S+1)
  – Control hazard => we fetched wrong instructions
    • Delaying them won’t solve anything, so they must be flushed
Like flushes, the “must” and the “can” differ
- No real **must**: we can avoid hazards by flushing
- But we **want to** stall if that can avoid a flush
- And we **can** stall whenever it’s convenient
  - Must still ensure forward progress!

Stalling to handle data dependences
- Simplest (and slowest) approach:
  - Stall read-reg stage until nothing remains in later stages
  - With 2-stage, stall stage 1 if a non-NOP is in stage 2
- Faster but more complex approaches
  - Stall until no register-writing instruction remains in later stages
  - Stall until no inst that writer to my src registers remains in later stages
  - Stall until forwarding can get us the values we need
Stalling in Verilog code

• For a pipeline FF between some stages A and M:
  always @(posedge clk or negedge reset)
    if(reset)
      wrreg_M<=1’b0;
    else if(flush_A)
      wrreg_M<=1’b0;
    else if(!stall_A)
      wrreg_M<=wrreg_A;
  – Note 1: if stalling stage X, must also stall stages before it
  – Note 2: when stalling fetch stage, don’t let PC change!
How to do Project 3

• Get it working with NOPs in the code
  – Change code to add NOPs in the right places
  – Note: “right” places will change with pipeline depth
  – This gets you 30 points

• Get it working with “heavy” stalls and flushing
  – Must run with original code (no NOPs added in .a32)
  – With “flush K” support in the pipeline: +20 points
  – More points if you use stalls to make it faster

• Then try to use smarter stalls and flushing
  – Very little of this will get you the other 50 points
With two stages (F and M):

\[
\text{assign stall}_F = \text{wrreg}_M \land (\text{wregno}_M = \text{rregno1}_F \lor \text{wregno}_M = \text{rregno2}_F);
\]
Smart stalling with more stages

• Which stage to stall?
  – The first stage where hazard makes us do something wrong that we won’t fix later
  – With two stages, this is first stage
    • We read wrong value from regs, and we use that wrong value in ALU

• With five stages w/o forwarding, this is reg-read
  – Wrong value from reg, must stall to read again

• With five stages w/ forwarding?
  – Reading wrong reg value is OK, forwarding fixes that
  – But if we forward the wrong value stall the stage in which we do forwarding!
Staling >1 stage

• If we stall stage X, must also stall stages 1..X-1
• Depending on what is done in which stage, different hazards might stall different stages
• In general, with stages A,B,C,etc.:
  assign stallto_A=<when to stall only A stage>;
  assign stallto_B=<when to stall up to stage B>;
  assign stallto_C=<when to stall up to stage C>;

  ...

 assign stall_A=stallto_A||stall_B;
 assign stall_B=stallto_B||stall_C;

  ...

This is in your hazard detection logic

Use these in the actual code that stalls pipeline-FF writes