U. Wisconsin CS/ECE 552
Introduction to Computer Architecture

Prof. Karu Sankaralingam

Pipelining (Chapter 6)

www.cs.wisc.edu/~karu/courses/cs552

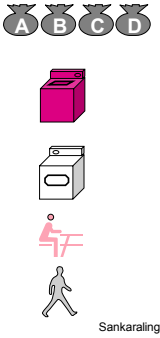
Slides combined and enhanced by Karu Sankaralingam from work by Falsafi, Hill, Marculescu, Nagle, Patterson, Roth, Rutenbar,Schmidt, Shen, Sohi, Sorin, Thottethodi, Vijaykumar, & Wood

Outline

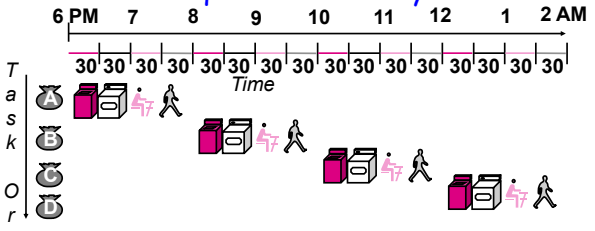
- Pipelining
  - What? Basic concepts
    - Overlapping execution
    - Latency vs. throughput
  - Why? Performance implications
    - Speedup
    - CPI, cycletime
  - How? Implementation challenges

Pipelining

- Laundry Example
- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
- Washer takes 30 minutes
- Dryer takes 30 minutes
- "Folder" takes 30 minutes
- "Stasher" takes 30 minutes to put clothes into drawers

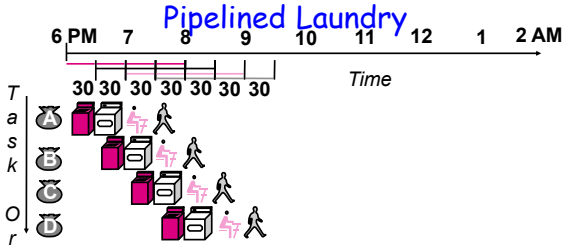


Sequential Laundry



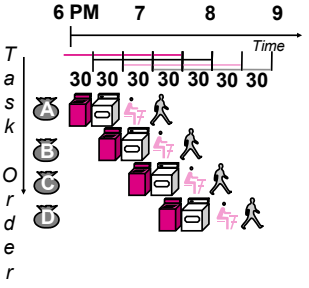
- Sequential laundry takes 8 hours for 4 loads
- If they learned pipelining, how long would laundry take?

Pipelined Laundry



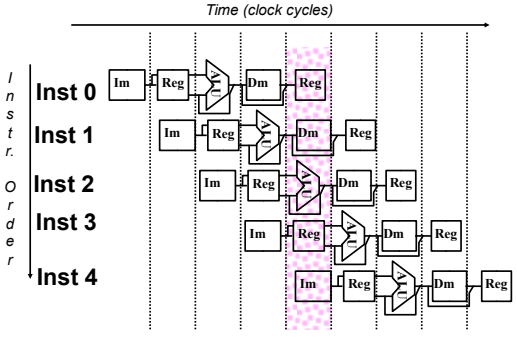
- Pipelined laundry takes 3.5 hours for 4 loads!
- Not esoteric computer architecture concept
- Natural, Intuitive

Pipelining lessons

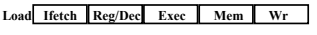


- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipe stages
- Pipeline rate limited by slowest pipeline stage
- Unbalanced lengths of pipe stages reduces speedup
- Time to "fill" pipeline and time to "drain" it reduces speedup
- Stall for Dependences

Seek to Pipeline Instructions

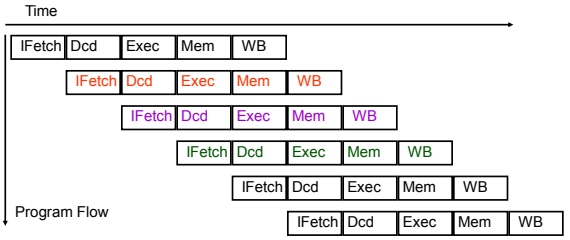


The Stages of Load



- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec: Calculate the memory address
- Mem: Read the data from the Data Memory
- Wr: Write the data back to the register file

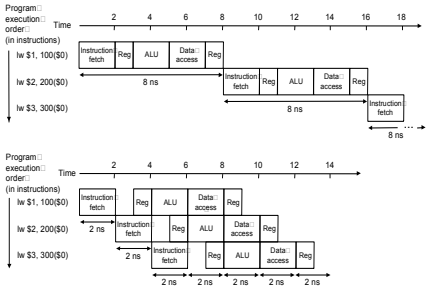
Pipelined Execution Representation



- Ideal speedup = 5, but …

Pipelining

- Improve perfomance by increasing instruction throughput



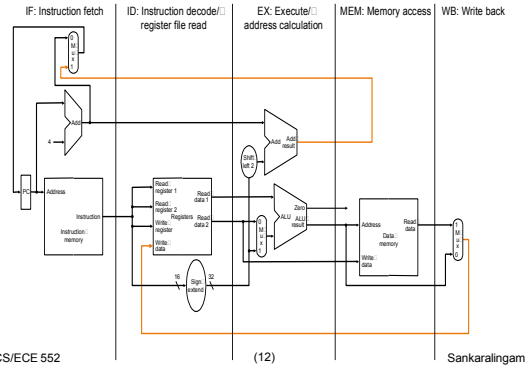
Ideal speedup is number of stages in the pipeline. Do we achieve this?

Non-uniform stages

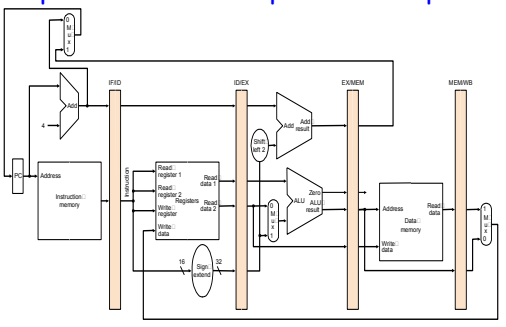
Maximum Speedup ≤ Number of stages

$$\text{Speedup} \leq \frac{\text{Time for unpipelined operation}}{\text{Time for longest stage}}$$

Pipeline Forecast: Single-Cycle Datapath

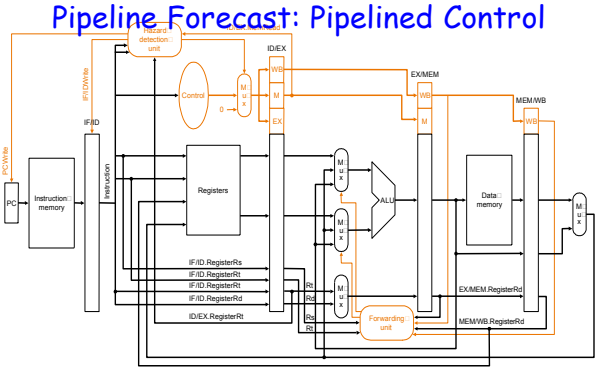


Pipeline Forecast: Pipelined Datapath



- Pipeline datapath with registers

Pipeline Forecast: Pipelined Control



Pipeline Forecast: Big Picture

- Datapath similar to single-cycle datapath
- Partition datapath with pipeline latches (D-FFs)
- Naïve Control
  - Generate single-cycle control signals
  - Pass control signals through pipeline latches
  - Apply control signals at appropriate stage/cycle
- Truth is more complex (instruction interact)

Instructions vs. Laundry

- Definitely:
  - Need to maintain "illusion" of sequential execution
  - Execution is actually overlapped.
- Pipeline Hazards
  - structural hazards: attempt to use the same resource two different ways at the same time
    - E.g., combined washer/dryer would be a structural hazard or folder busy doing something else (watching TV)
  - data hazards: attempt to use item before it is ready
    - E.g., one sock of pair in dryer and one in washer; can't fold until get sock from washer through dryer
    - instruction depends on result of prior instruction still in the pipeline
  - control hazards: attempt to make a decision before condition is evaulated
    - E.g., washing football uniforms and need to get proper detergent level; need to see after dryer before next load in
    - branch instructions

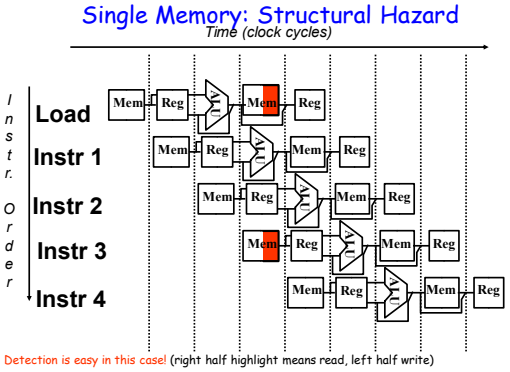
Hazards

- Structural hazards
  - Two instructions need the same hardware
- Data Hazards
  - Data not ready
- Control Hazards
  - Which instruction to fetch? Not known.

Hazards

- Can always resolve hazards by waiting
  - pipeline control must detect the hazard
  - take action (or delay action) to resolve hazards
- Delays
  - Pipeline stalls/bubbles
  - Reduce speedup

Single Memory: Structural Hazard



Detection is easy in this case! (right half highlight means read, left half write)

Structural Hazards

- If 1.3 memory accesses per instruction
  - How?
  - 1 per instruction for instruction fetch
  - Fraction for data load/store
    - Depends on instruction mix
    - 20% load + 10% store
    - 15% load + 15% store
- CPI is atleast 1.3 (otherwise memory is used more than 100%)

Data Hazards

add r1 ,r2,r3

sub r4, r1 ,r3

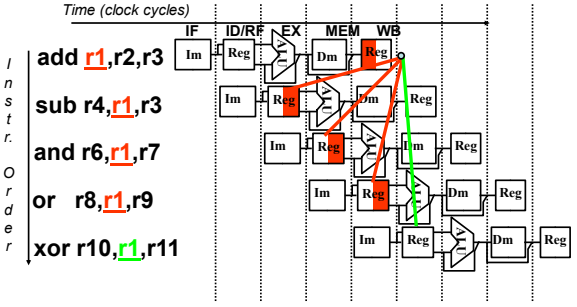
and r6, r1 ,r7

or r8, r1 ,r9

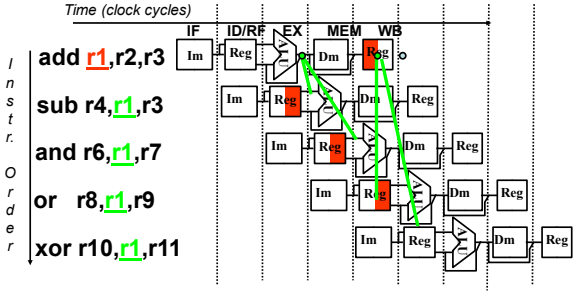
xor r10, r1 ,r11

Hazards on r1

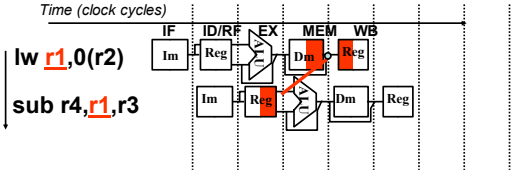
- Dependencies backwards in time



Data Hazard Solution



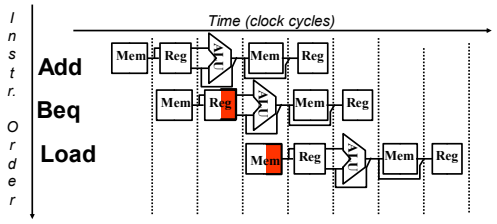
Forwarding (a.k.a. bypassing)



- Can't solve with forwarding:
  - Must delay/stall instruction dependent on loads

Control Hazard: Solutions

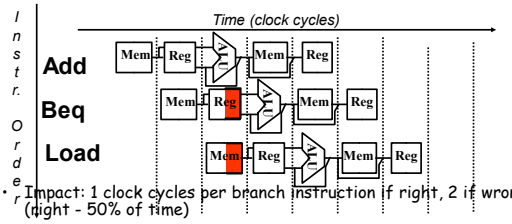
- Stall: wait until decision is clear
  - Its possible to move up decision to 2nd stage by adding hardware to check registers as being read



- Impact: 2 clock cycles per branch instruction => slow

Control Hazard: Solutions

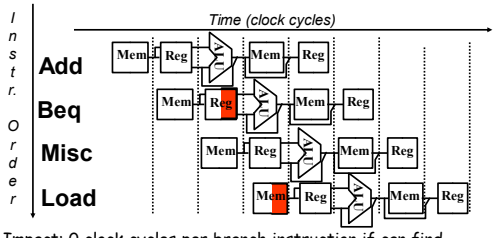
- Predict: guess one direction then back up if wrong
  - Predict not taken



- Impact: 1 clock cycles per branch instruction if right, 2 if wrong (right - 50% of time)
- More dynamic scheme: history of 1 branch (- 90%)

Control Hazard: Solutions

- Redefine branch behavior (takes place after next instruction) "delayed branch"



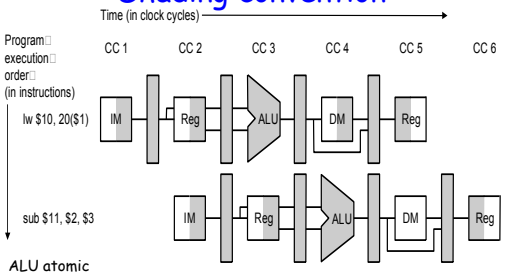
- Impact: 0 clock cycles per branch instruction if can find instruction to put in "slot" (- 50% of time)
- As launch more instruction per clock cycle, less useful

STOPPED HERE

Pipelined Processor Design

- Designing a pipelined processor
  - Associate resources with states
  - Resources not necessarily atomic
    - Register reads and writes can happen in the same cycle
    - Writes in first half of cycle and reads in the second half
  - Assert appropriate controls in each stage
    - Make sure all necessary information is carried through inter-pipestage flip-flops

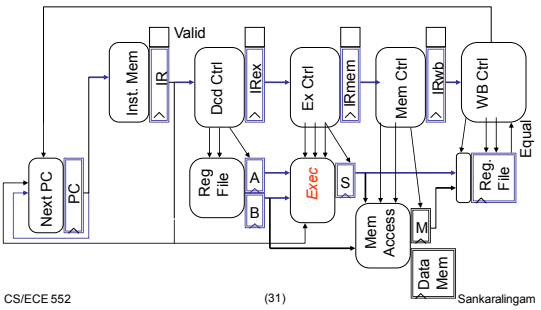
Shading convention



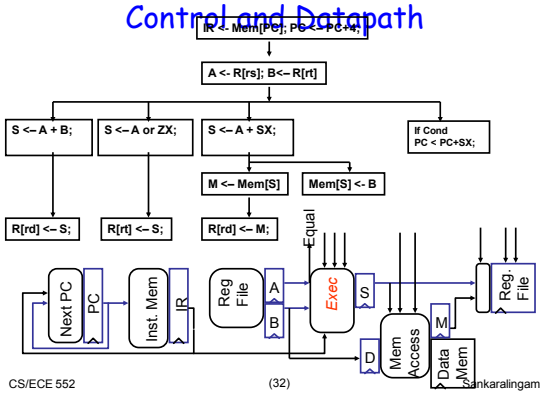
- ALU atomic
- Register file/memory can handle read/write in the same cycle
- Memory cannot handle two reads

Pipelined Processor (slide version)

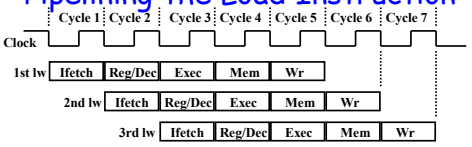
- What happens if we start a new instruction every cycle?



Control and Datapath

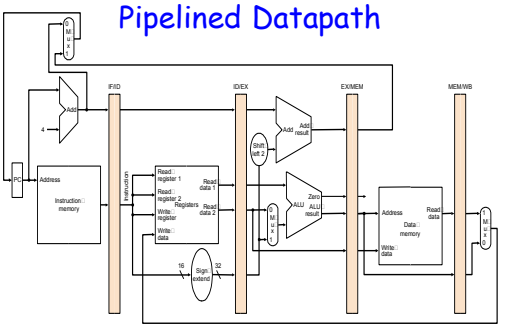


Pipelining the Load Instruction



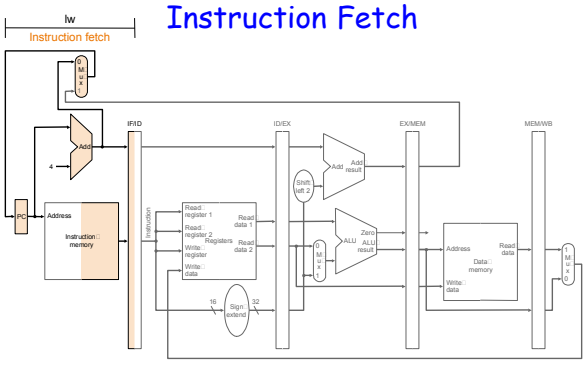
- The five independent functional units in the pipeline datapath are:
  - Instruction Memory for the Ifetch stage
  - Register File's Read ports (bus A and busB) for the Reg/Dec stage
  - ALU for the Exec stage
  - Data Memory for the Mem stage
  - Register File's Write port (bus W) for the Wr stage

Pipelined Datapath

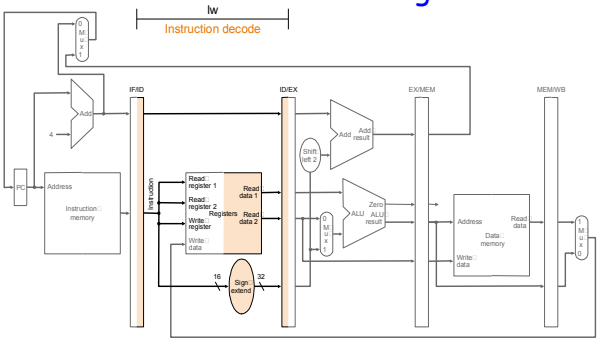


- Pipeline datapath with registers

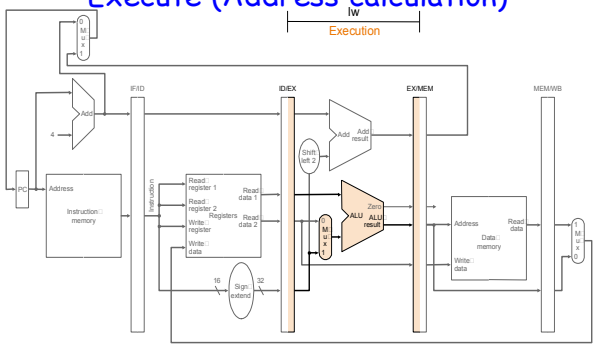
Instruction Fetch



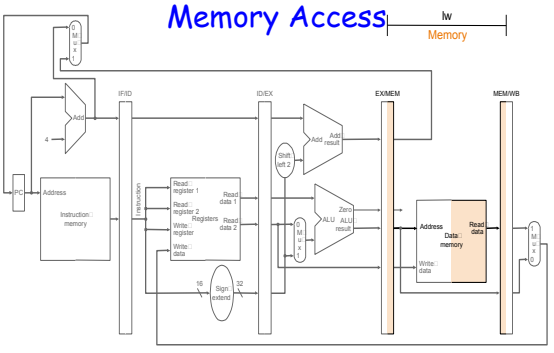
Instruction Decode/Reg Read



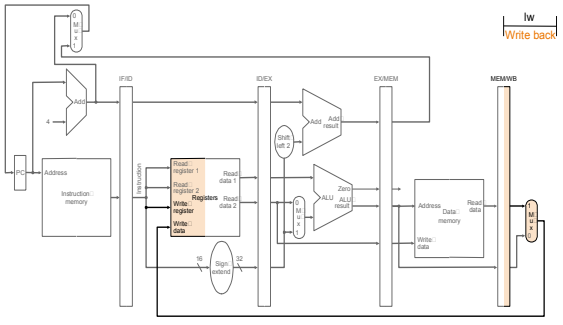
Execute (Address calculation)



Memory Access



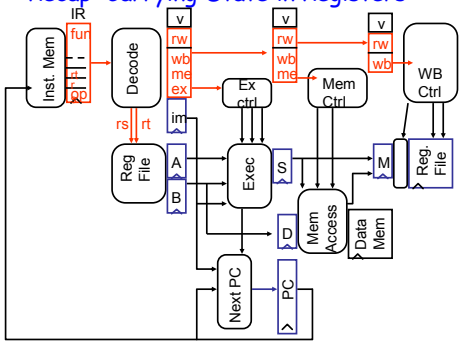
Writeback



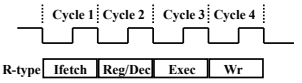
Pipeline registers

- Length of Pipeline registers
- Book says
  - IF/ID : IR (32), PC+4 (32) : 64 bits
  - ID/EX: IR (32), PC+4 (32) + RegA + RegB : 128 bits
  - EX/MEM: ALUout(32) + zero(1) + PC+4+SX(imm) (32) : 97
  - MEM/WB: ALUout (32) + MemData(32) : 64
- Corrections:
  - ALUout and MemData
  - Destination register (5 bits)
  - Other control bits (IR not going through)

Recap: Carrying State in Registers

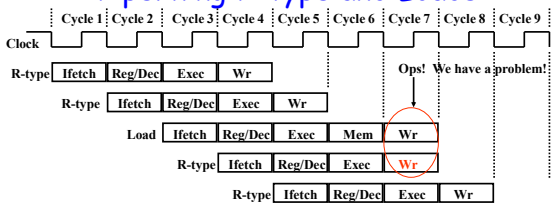


The Four Stages of R-type



- 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

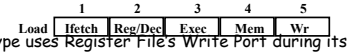
Pipelining R-type and Loads



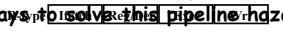
- We have pipeline conflict or structural hazard:
  - Two instructions try to write to the register file at the same time!
  - Only one write port

Key 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

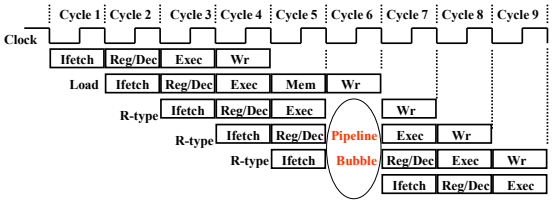


  - R-type uses Register File's Write Port during its 4th stage



- **2 ways to solve this pipeline hazard.**

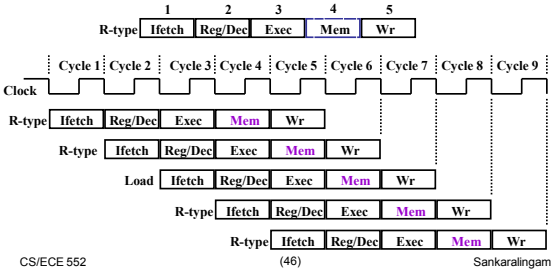
Soln.1: Insert "Bubble"



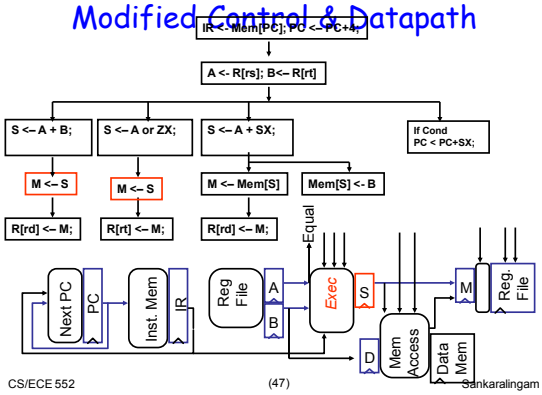
- Insert a "bubble" into the pipeline to prevent 2 writes at the same cycle
  - The control logic can be complex.
  - Lose instruction fetch and issue opportunity.
- No instruction is started in Cycle 6!

Soln.2: Delay R-type's Write

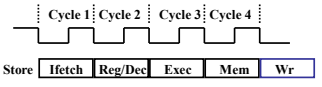
- Delay R-type's register write by one cycle:
  - Now R-type instructions also use Reg File's write port at Stage 5
  - Mem stage is a NOOP stage: nothing is being done.



Modified Control & Datapath

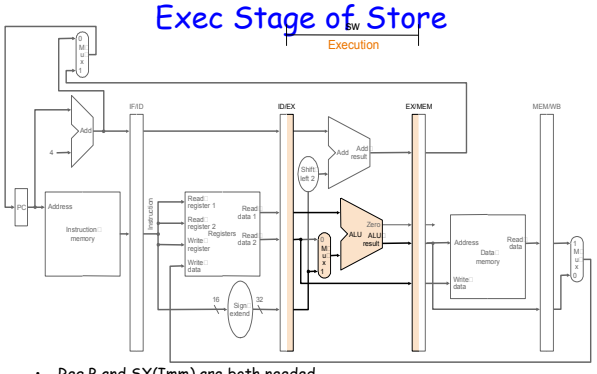


Four Stages of Store



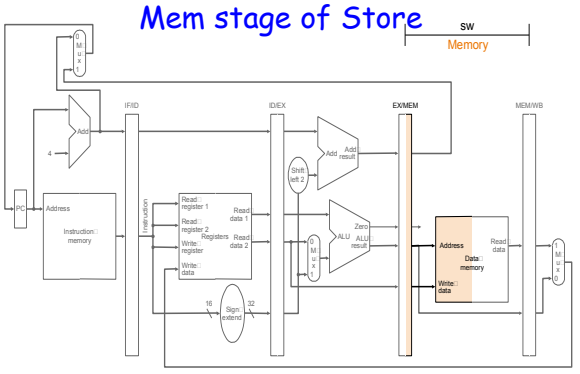
- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec: Calculate the memory address
- Mem: Write the data into the Data Memory

Exec Stage of Store

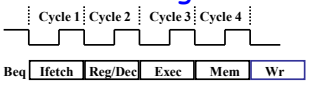


- Reg B and SX(Imm) are both needed

Mem stage of Store



Three Stages of Beq



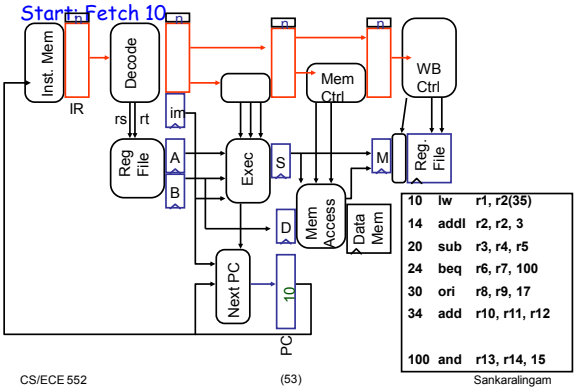
- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- Reg/Dec:
  - Registers Fetch and Instruction Decode
- Exec:
  - compares the two register operand,
  - select correct branch target address
  - latch into PC
- Four stages as in book
  - Assume one delay slot (shadow instruction)
  - One "predict not taken" instruction

Visualizing the pipeline

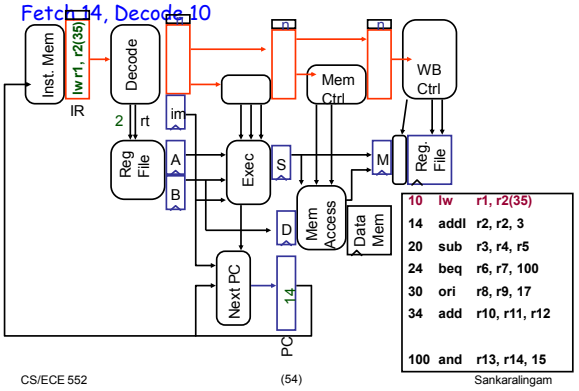
```
10    lw     r1, r2(35)
14    addl   r2, r2, 3
20    sub    r3, r4, r5
24    beq    r6, r7, 100
30    ori    r8, r9, 17
34    add    r10, r11, r12

100   and    r13, r14, 15
```

these addresses are octal

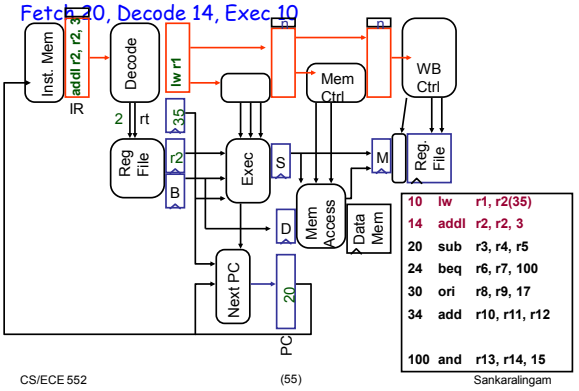
Start: Fetch 10



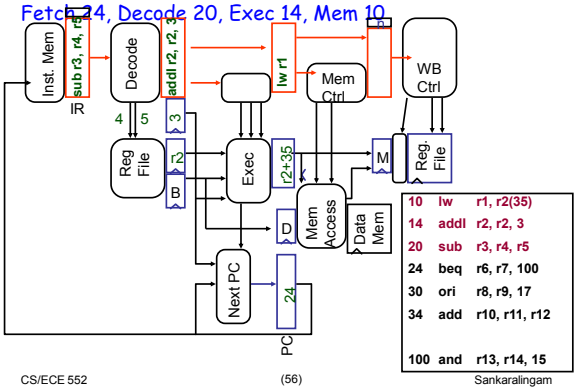
Fetch 14, Decode 10

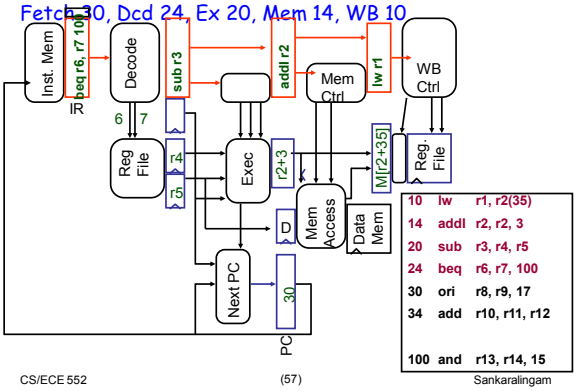


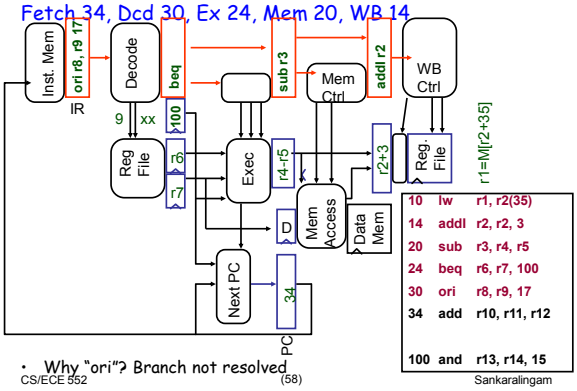
Fetch 20, Decode 14, Exec 10



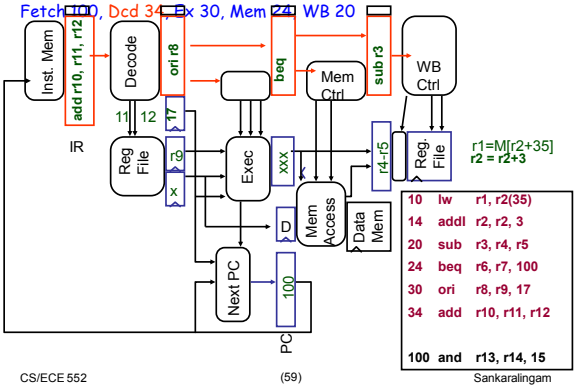
Fetch 24, Decode 20, Exec 14, Mem 10

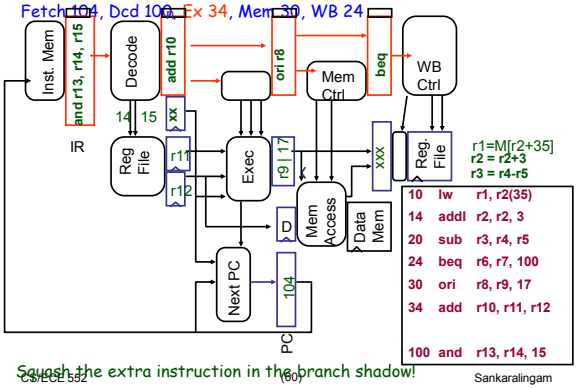






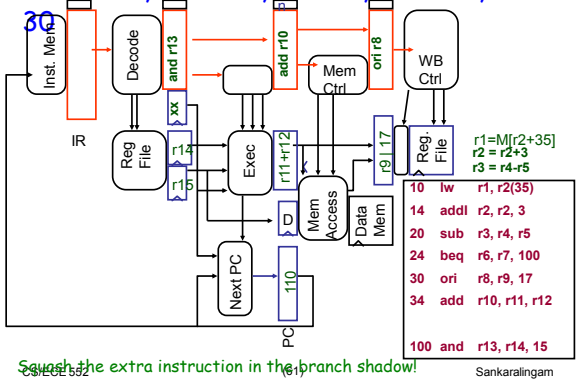
- Why "ori"? Branch not resolved





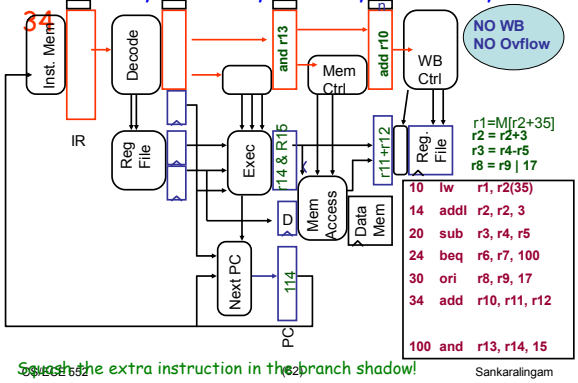
Squash the extra instruction in the branch shadow!

Fetch 110, Dcd 104, Ex 100, Mem 34, WB 30



Squash the extra instruction in the branch shadow!

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

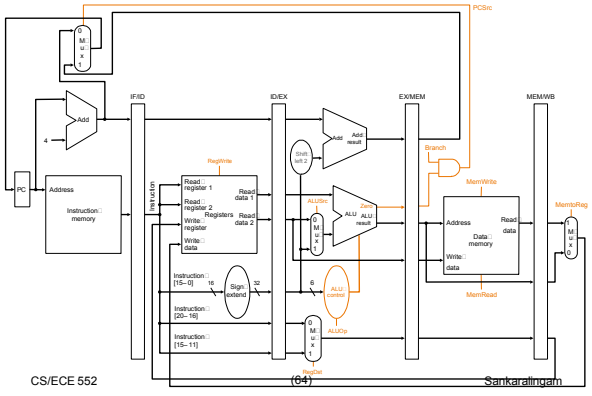


Squash the extra instruction in the branch shadow!

Outline

- Pipeline Datapath
  - Under simplifying assumptions
    - All independent instructions
    - Talked about bypassing/forwarding
      - Datapath not capable of handling forwarding yet
    - Walk-through
    - Delayed branches
- Control
  - Most complicated (read irregular/unstructured)
  - Gets more complicated for pipelined processors
    - But we'll start with a simple case

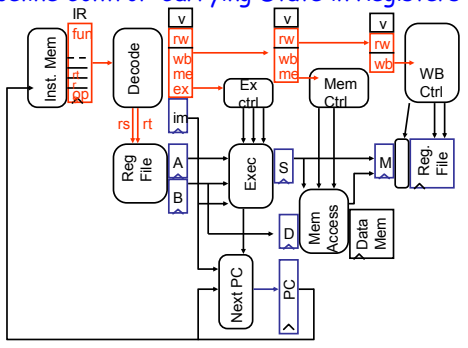
Pipelined Datapath with Controls



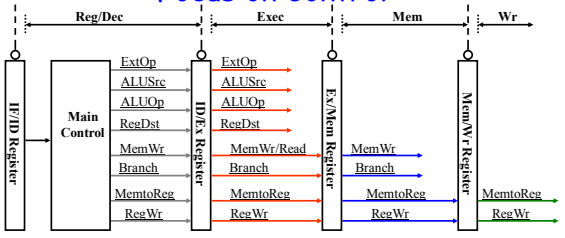
Pipeline Control vs. Single cycle control

- Similarity
  - Replicated functional units like single-cycle implementation
    - Imem and Dmem
    - Separate adder for PC+SX(Imm) computation
- What about
  - PCWrite? IR-write?
  - Write enable for the pipeline registers?

Pipeline Control: Carrying State in Registers



Focus on Control



- The Main Control generates the control signals during Reg/Dec
  - Control signals for Exec (ExtOp, ALUSrc, ...) are used 1 cycle later
  - Control signals for Mem (MemWr Branch) are used 2 cycles later
  - Control signals for Wr (MemtoReg MemWr) are used 3 cycles later

Generating controls

- We just simplified the problem
  - Reduced pipeline control to single-cycle control (Almost)
  - Generate controls once
  - Consume (i.e., use and discard) signals as you proceed along the pipeline stages
- Identify Stage of consumption for all control signals

Meaning of controls

- RegWr : 1-> write, 0-> no write
- MemToReg : 1-> MDR, 0-> ALUOut
- RegDst : 1->rd, 0->rt
- ALUOp<1:0> : 00->Add,01->Sub,10->'funct'
- ALUSrc : 0->RegB, 1->SX(Imm)
- ExtOp**: --- needed for ORI---
- Branch: 0->non-branch inst, 1-> branch
- MemRead: 1->memread, 0-> no memread
- MemWrite: 1->memwrite, 0->no memwrite
- ALU control abstracted away (as before)
  - Inputs: ALUop (2 bits), 6 "funct" bits from IR

WB <1:0>

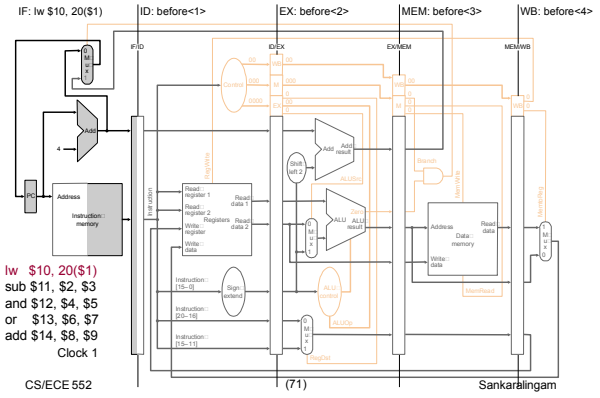
Exec <3:0>

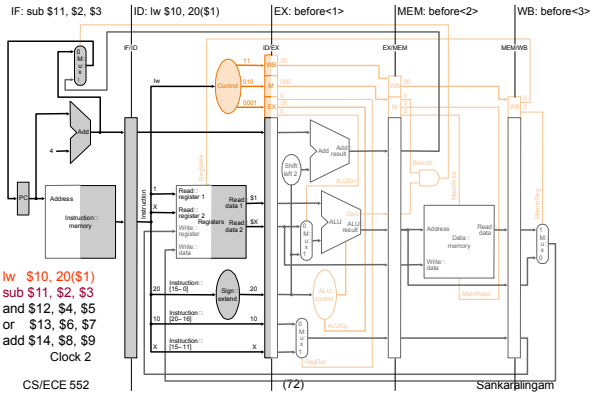
Mem <2:0>

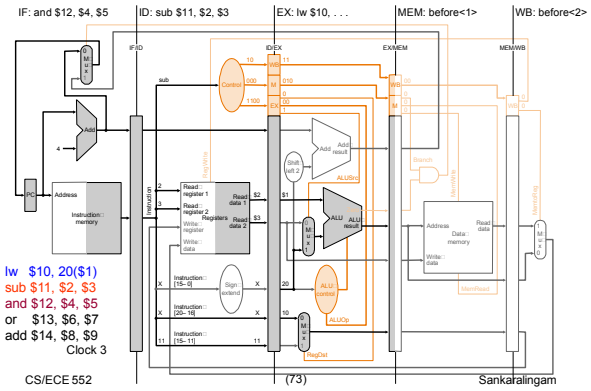
Pipeline Walkthrough with controls

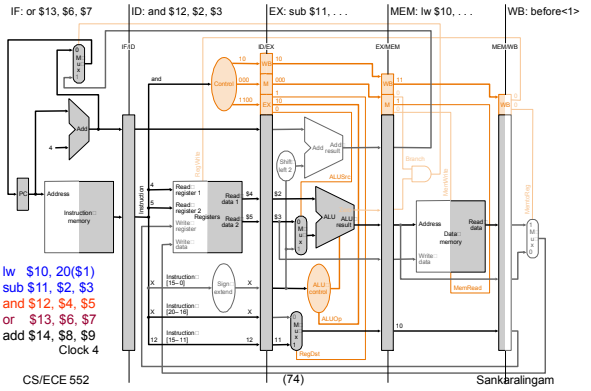
- Use walkthrough worksheets
  - Use code segment shown
  - Fill in controls
  - Interesting stages
    - Controls generated in Decode stage
    - Controls consumed in subsequent stages

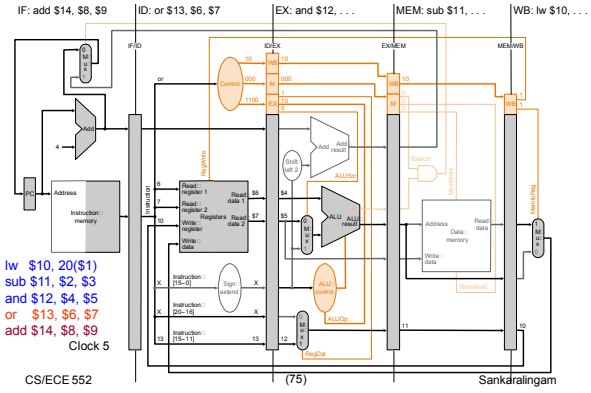
```
lw   $10, 20($1)
sub $11, $2, $3
and $12, $4, $5
or    $13, $6, $7
add $14, $8, $9
```

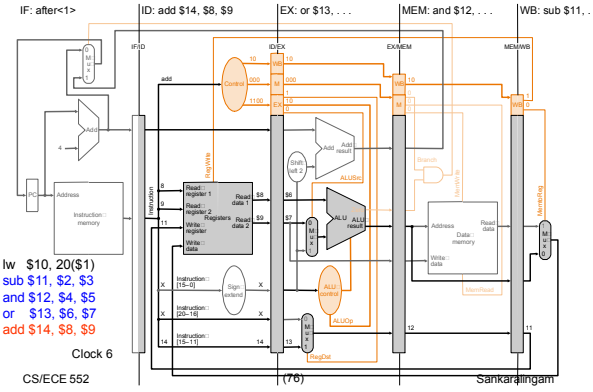



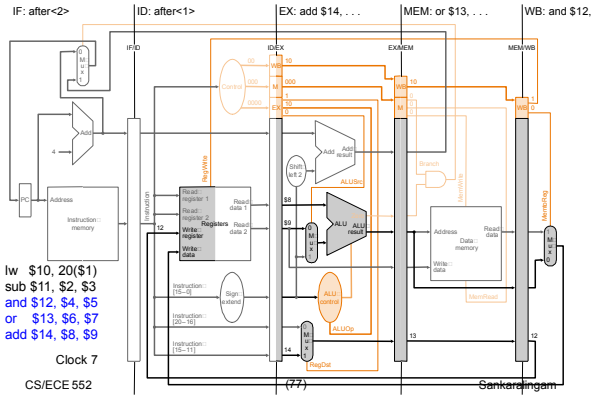


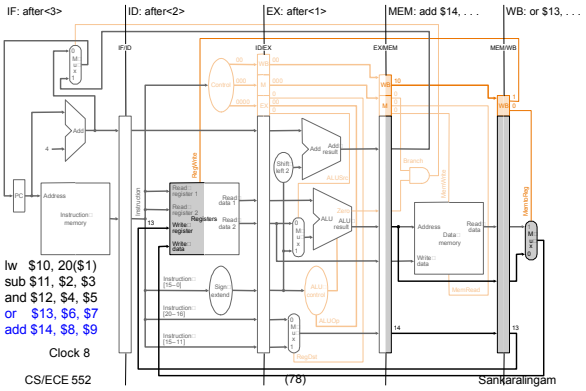

IF: and $12, $4, $5
ID: sub $11, $2, $3
EX: lw $10, . . .
MEM: before<1>
WB: before<2>
lw $10, 20($1)
sub $11, $2, $3
and $12, $4, $5
or $13, $6, $7
add $14, $8, $9
Clock 3
CS/ECE 552
(73)
Sankaralingam

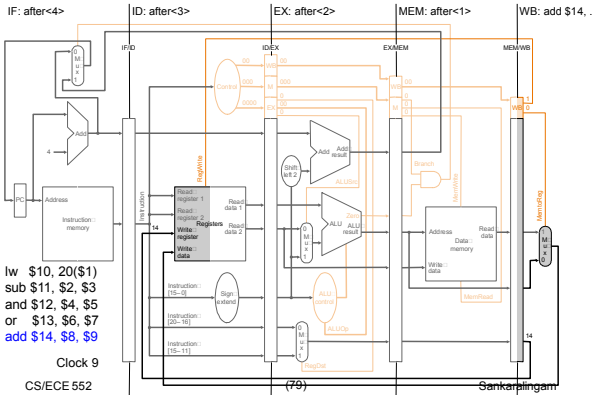

IF: or $13, $6, $7
ID: and $12, $2, $3
EX: sub $11, . . .
MEM: lw $10, . . .
WB: before<1>
lw $10, 20($1)
sub $11, $2, $3
and $12, $4, $5
or $13, $6, $7
add $14, $8, $9
Clock 4
CS/ECE 552
(74)
Sankaralingam


IF: add $14, $8, $9
ID: or $13, $6, $7
EX: and $12, . . .
MEM: sub $11, . . .
WB: lw $10, . . .
lw $10, 20($1)
sub $11, $2, $3
and $12, $4, $5
or $13, $6, $7
add $14, $8, $9
Clock 5
CS/ECE 552
(75)
Sankaralingam


IF: after<1>
ID: add $14, $8, $9
EX: or $13, . . .
MEM: and $12, . . .
WB: sub $11, .
lw $10, 20($1)
sub $11, $2, $3
and $12, $4, $5
or $13, $6, $7
add $14, $8, $9
Clock 6
CS/ECE 552
(76)
Sankaralingam







Implementing Pipeline control

- How do we design the control logic block?
  - Similar to single-cycle implementation
  - Derive logic expressions
    - E.g. MemtoReg = lw
    - ALUSrc = lw OR sw
    - RegWrite = R-type OR lw
  - Implement Combinational logic
    - PLA implementation
    - ROM implementation

Pipeline registers

- Preliminary estimates
  - IF/ID : IR (32), PC+4 (32) : 64 bits
  - ID/EX: IR (32), PC+4 (32) + RegA + RegB : 128 bits
  - EX/MEM: ALUout(32) + zero(1) + PC+4+SX(imm) (32) : 97
  - MEM/WB: ALUout (32) + MemData(32) : 64
- Corrections:
  - ALUout and MemData
  - Destination register (5 bits)
  - Other control bits (IR not going through)

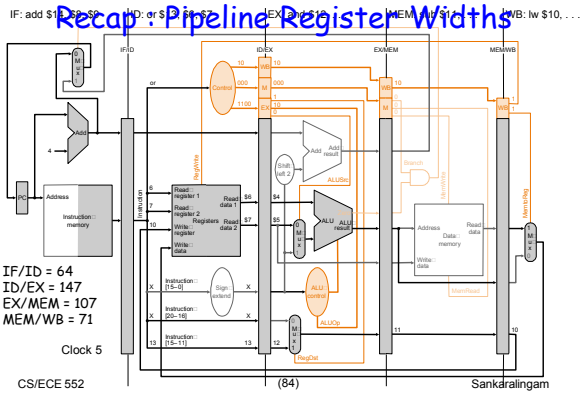
Implementing Pipeline Control

- Exercise
  - Compute required bit-width of pipeline registers
  - Before the next lecture (NOT Feb 23rd)

Summary

- Only slightly more complicated than single cycle
  - not really, only because we:
    - ignored complications of forwarding, branch prediction, pipeline bubbles, squashing mispredicted instructions (after branches)
  - Need deeper understanding of hazards
    - need to modify datapath as well

Recap : Pipeline Register Widths

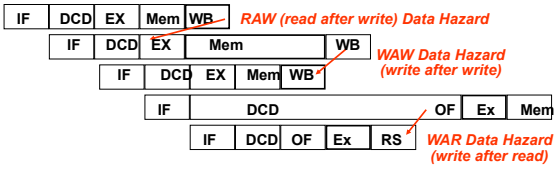


Next

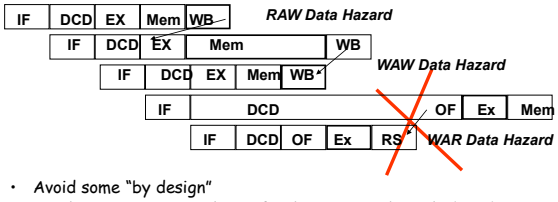
- Pipeline datapath and control assuming independent instructions (no hazards)
- Data hazards
  - Types
  - Detecting RAW hazards
  - Handling RAW hazards (Partial)
    - Datapath
    - Control behavior

Data Hazards

- Challenge: maintain illusion of sequential execution
- Types of data hazards
  - RAW, WAR, WAW



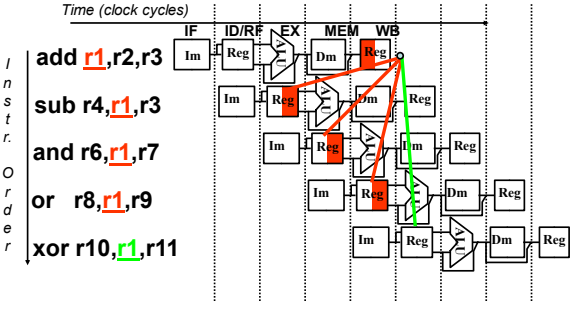
Data Hazards



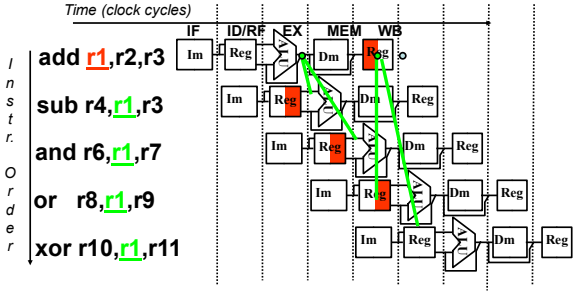
- Avoid some "by design"
  - eliminate WAR by always fetching operands early (DCD) in pipe
  - eleminate WAW by doing all WBs in order (last stage, static)
- Detect and resolve remaining ones
  - stall or forward (if possible)

Hazards on r1

- Dependencies backwards in time



Data Hazard Solution



Handling RAW Hazards

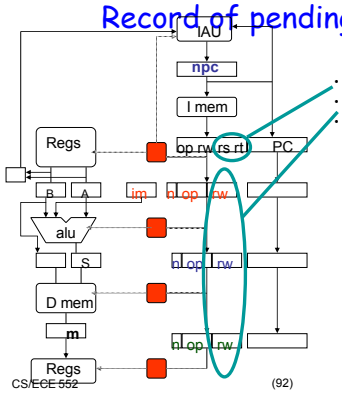
- Pre-requisite for handling RAW hazard
  - Detection!
  - Need to know:
    - Pending writes
      - available results that haven't been written back to registers
    - Operand Reads
      - Later instructions that potentially use these values
  - Instructions may not write to register file (store, branch)

Detecting RAW hazards

- Suppose instruction i is about to be issued and a predecessor instruction j is in the instruction pipeline.
- A RAW hazard exists on register ρ if $\rho \in Rregs(i) \cap Wregs(j)$
  - Keep a record of pending writes (for inst's in the pipe) and compare with operand regs of current instruction.
  - When instruction issues, reserve its result register.
  - When on operation completes, remove its write reservation.

- A WAW hazard exists on register ρ if $\rho \in Wregs(i) \cap Wregs(j)$
- A WAR hazard exists on register ρ if $\rho \in Wregs(i) \cap Rregs(j)$

Record of pending writes



- Current operand registers
- Pending writes
- hazard <=

((rs == rw_{ex}) & $regW_{ex}$)
OR
((rs == rw_{mem}) & $regW_{me}$)
OR
((rs == rw_{wb}) & $regW_{wb}$)
OR
((rt == rw_{ex}) & $regW_{ex}$)
OR
((rt == rw_{mem}) & $regW_{me}$)
OR
((rt == rw_{wb}) & $regW_{wb}$)

Logic equations for Hazard Detection

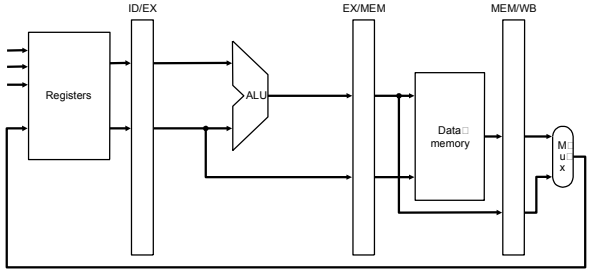
- Restatement of equations
- Text book version
  - WB stage is not really a hazard
    - Data is written in first half of cycle, read in 2nd half
  - EX/MEM.RegisterRd = ID/EX.RegisterRs
  - EX/MEM.RegisterRd = ID/EX.RegisterRt
  - MEM/WB.RegisterRd = ID/EX.RegisterRs
  - MEM/WB.RegisterRd = ID/EX.RegisterRt

Lookahead: Forwarding datapath

- We know how to detect RAW hazards
- Now,
  - Modify Datapath to enable forwarding
  - Desired control behavior

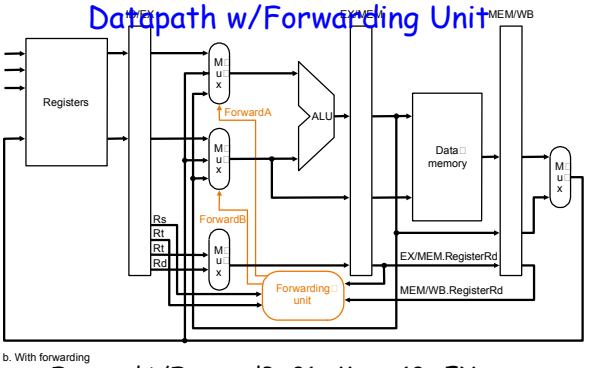
Base Pipelined Datapath

- Simplified representation of pipelined datapath
  - To avoid clutter



a. No forwarding

Datapath w/Forwarding Unit



b. With forwarding

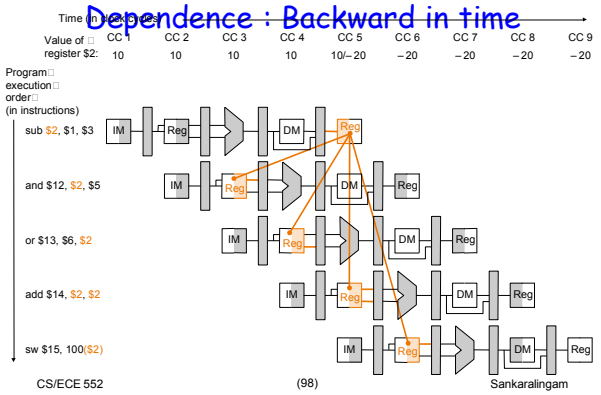
- ForwardA/ForwardB: 01->Mem, 10->EX

Data Hazards and Forwarding: Walkthrough

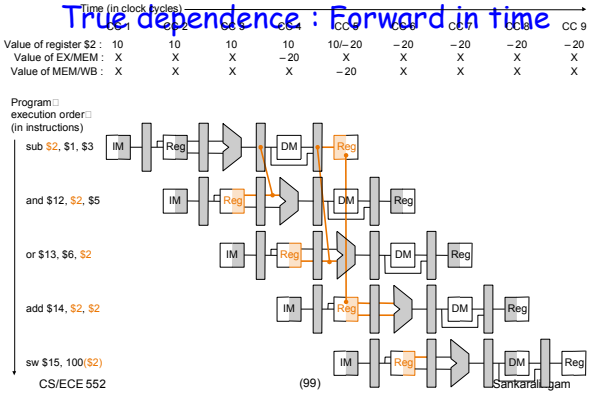
- Code snippet
  - identify hazards
  - identify forwarding paths

sub $2, $1, $3
and $4, $2, $5
or $4, $4, $2
add $9, $4, $2

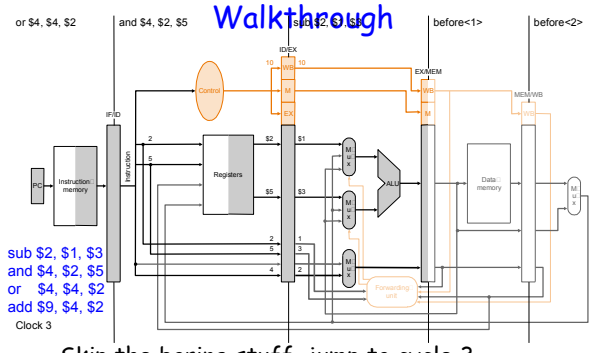
Dependence : Backward in time



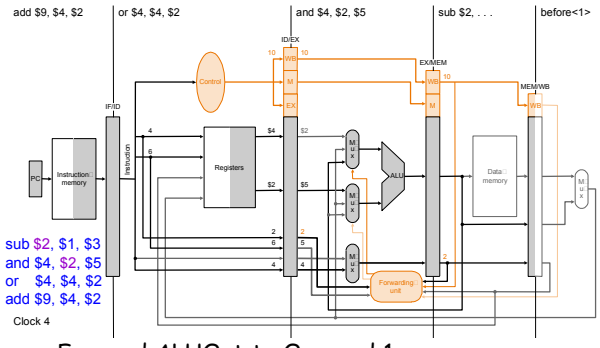
True dependence : Forward in time



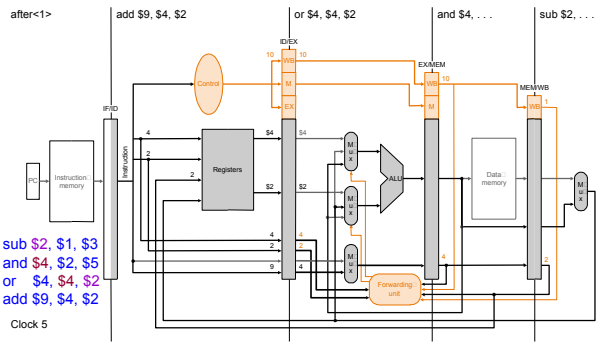
Walkthrough



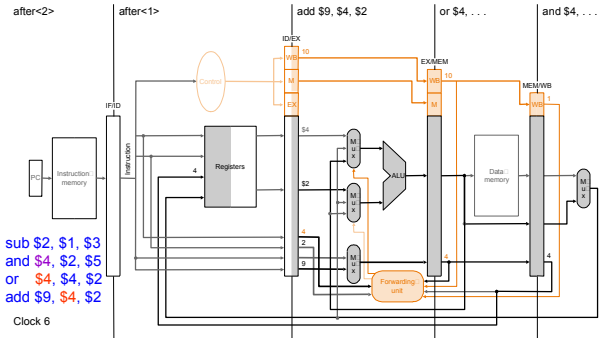
- Skip the boring stuff, jump to cycle 3



- Forward ALUOut to Operand 1

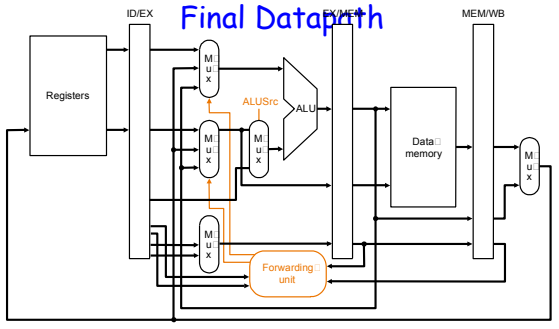


- Forward ALUout to Op1, Mem to Op2



- Two candidates match, forward the latest

Final Datapath



- "Imm" can be 2nd operand

Forwarding Control Behavior

- EX hazard

```
If (EX/MEM.RegWrite AND      // not store or branch
      EX/MEM.RegsterRd != 0   AND  // Result is used
      EX/MEM.RegisterRd = ID/EX.RegisterRs)
      ForwardA = 10

If (EX/MEM.RegWrite AND
      EX/MEM.RegsterRd != 0   AND
      EX/MEM.RegisterRd = ID/EX.RegisterRt)
      ForwardB = 10
```

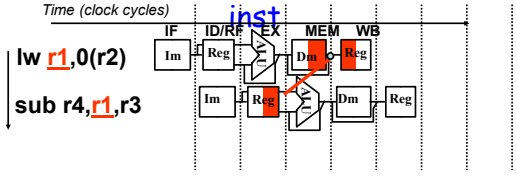

Forwarding Control Behavior

- MEM hazard

```
If (MEM/WB.RegWrite AND
      MEM/WB.RegsterRd != 0   AND
      MEM/WB.RegisterRd = ID/EX.RegisterRs)
      ForwardA = 01

If (MEM/WB.RegWrite AND
      MEM/WB.RegsterRd != 0   AND
      MEM/WB.RegisterRd = ID/EX.RegisterRt)
      ForwardB = 01
```

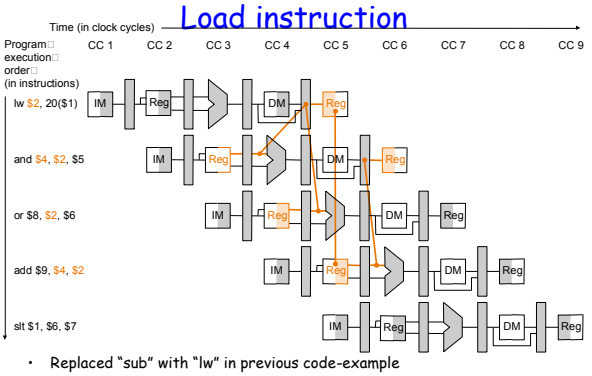
- Does this fully meet our requirements ?

Lookahead: RAW hazard with load inst



- Forwarding as solution to RAW hazard
  - possible if no (true) dependence going backwards in time
  - True for R-type instructions
    - Data available after EX stage (i.e., at ALUOut)
  - Not true for load instruction

Load instruction

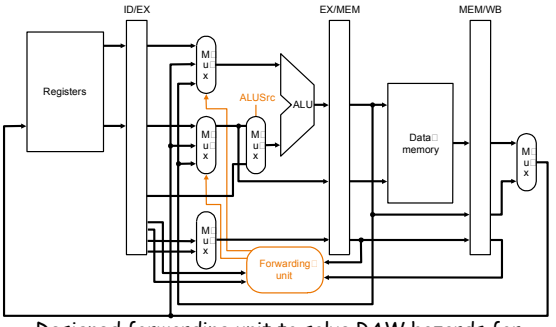


- Replaced "sub" with "lw" in previous code-example

Solution

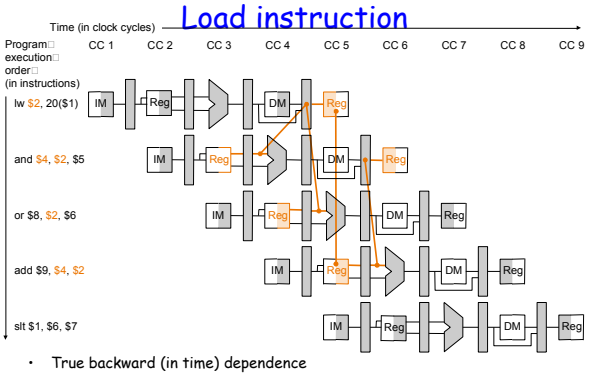
- Catch-all solution for hazards
  - Stall
    - always works, but hurts performance
    - Use as last resort
- Challenge:
  - Modify pipeline implementation to support stalls when hazards are detected

Recap



- Designed forwarding unit to solve RAW hazards for R-type instructions

Load instruction



- True backward (in time) dependence

Hazards with load instruction

- True dependencies: backward in time
- Stall the pipeline
- Minor change in terminology
  - If forwarding can solve it, it is not a hazard!
  - "Hazard" refers only to true backward dependencies in time.

Handling the hazard

- As before
  - Detection
    - Logic equations to detect hazard
  - Actual stalling
    - Datapath/control modifications to achieve stalling

Detection

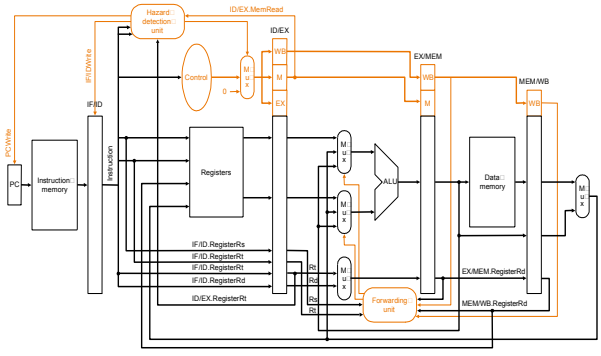
- Conditions
  - Preceding instruction must read memory
    - MemRead must be asserted
  - Destination of preceding instruction (rt) must be one of operands of current instruction
- Logic equations– restate above conditions formally
  - If(ID/EX.MemRead AND
    ((ID/EX.RegRt = IF/ID.RegRs) OR
    (ID/EX.RegRt = IF/ID.RegRt)))
    STALL

```
lw   $2, 20($1)
and $4, $2, $5
or   $4, $4, $2
add $9, $4, $2
```

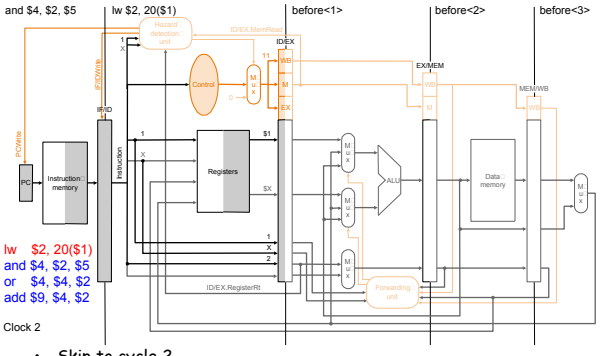

Stalling the pipeline

- Instruction cannot proceed
  - Following instruction must be stalled too.
  - Otherwise state in pipeline registers is overwritten
- Preceding instructions may proceed as usual
- Solution
  - inject NOP into EX/Mem pipeline
  - Prevent writes to PC to IF/ID register

Datapath

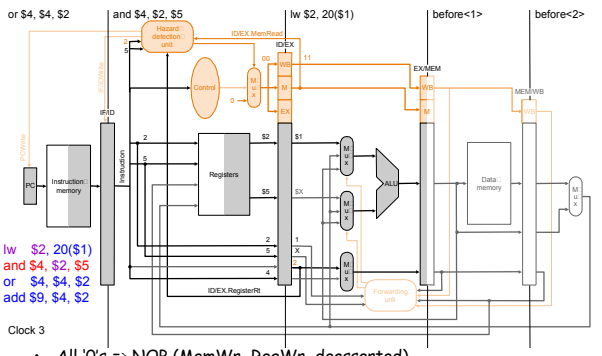


Walk-through (1 of 6)



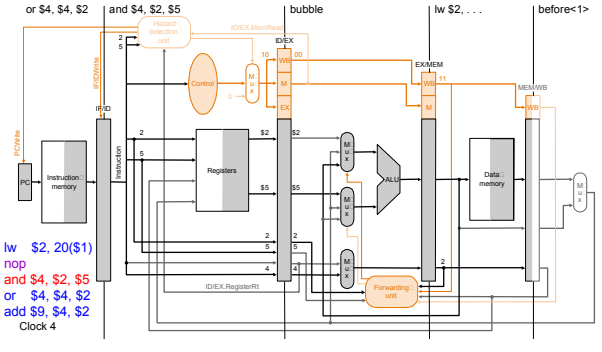
- Skip to cycle 2

Walk-through (2 of 6)

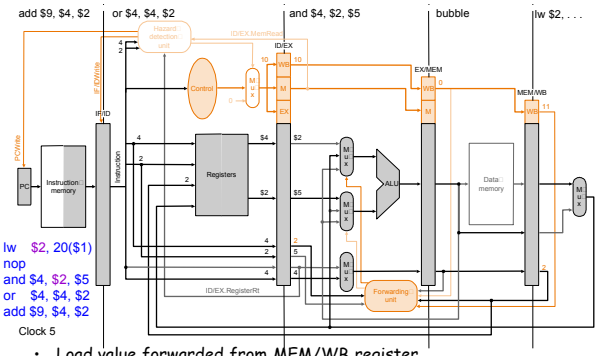


- All '0's => NOP (MemWr, RegWr, deasserted)

Walk-through (3 of 6)

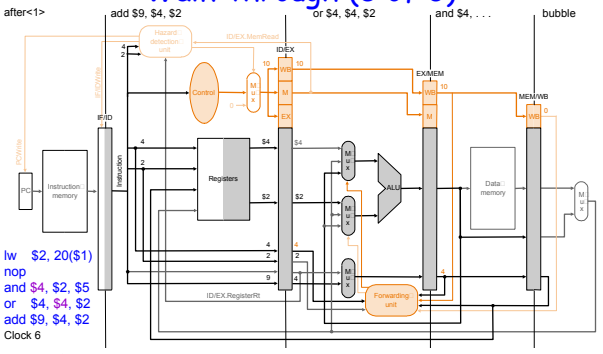


Walk-through (4 of 6)



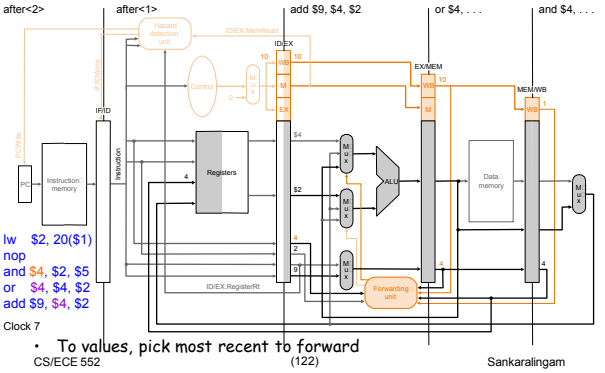
- Load value forwarded from MEM/WB register

Walk-through (5 of 6)



- $4 value forwarded from EX/MEM register

Walk-through (6 of 6)

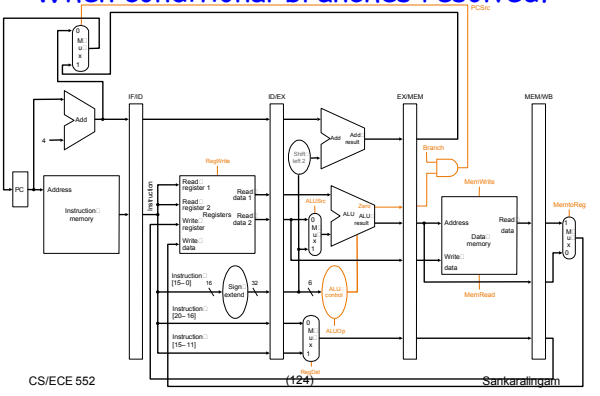


- To values, pick most recent to forward

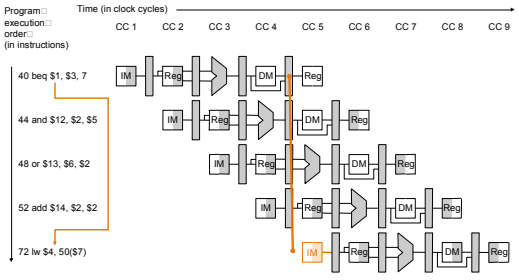
RAW Hazard with Loads: Summary

- True backward dependencies in time
  - Need to stall
- Stall achieved by
  - Detecting hazard (remember logic equation)
  - Inserting NOP (all EX/MEM/WB controls set to 0)
  - Preventing IF/ID register and PC from being overwritten
- Next Branch/Control Hazards

When conditional branches resolved?



Branch Hazards

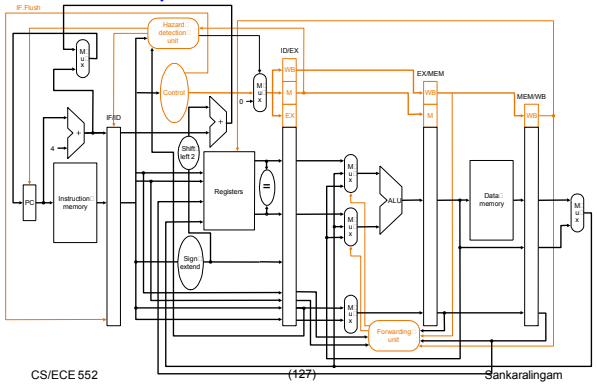


- Branch resolved in the MEM stage
- If taken,
  - PC<- PC + 4 + SX(Imm*4)
  - 40 + 4 + 7*4 = 72

Control/Branch Hazards

- Branch resolved in the MEM stage
  - But next instruction has to fetched in the next cycle
  - Reduce the penalty by moving decision earlier in pipeline
    - Need additional comparator (r1=r2?) and adder (PC+4+SX(IMM)*4)
  - Reduced penalty from 3 cycles to 1 cycle

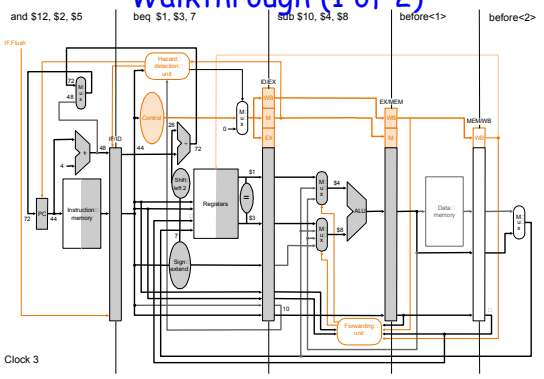
Datapath for branch hazards



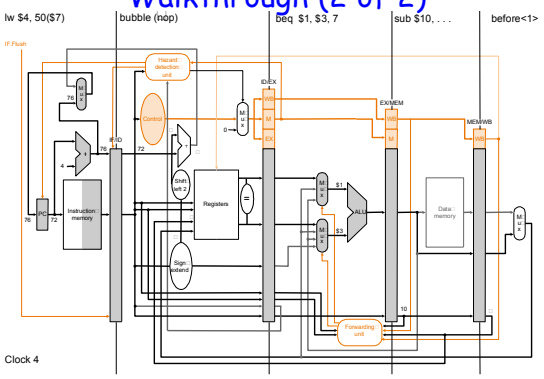
Eliminate 1-cycle stall?

- Two solutions
  - Predict branch is always not taken
    - More sophisticated prediction schemes
  - Delay slots
    - Compiler's problem
- Walkthrough example for solution #1
  - Predict not taken

Walkthrough (1 of 2)



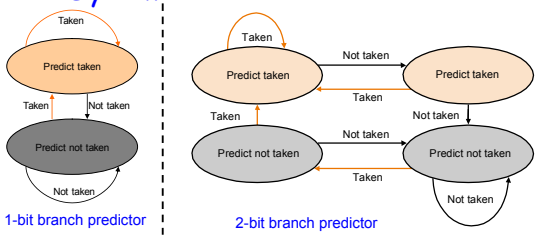
Walkthrough (2 of 2)



Dynamic Branch Prediction

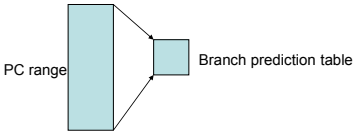
- Better than static prediction
  - Branches are predictable
  - ~90% of program execution time is spent in ~10% of code (inner loops)
  - Think of a program loop of N iterations
    - Taken N-1 times
    - Not taken last time

Dynamic Branch Prediction



- How does hardware "learn" branch behavior?
- Store each branch instruction's history ***
  - If a branch was taken "recently", predict taken
    - One bit saturating counter
    - Two bit counters

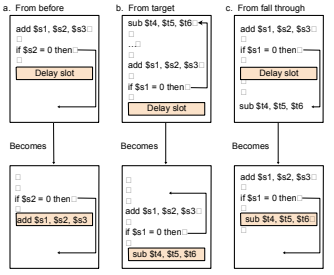
Branch Prediction



- Store each branch's history ***
  - Not really
- Keep a small table indexed by program counter
- PC is large (32 bit number)
- Mapping to number of table entries
  - E.g. 16-entry branch prediction table
  - Mapping: use last 4 bits of PC
- Problem: Multiple branches may map to same entry in table -- Aliasing

"Easy way"* to hide branch hazard delay

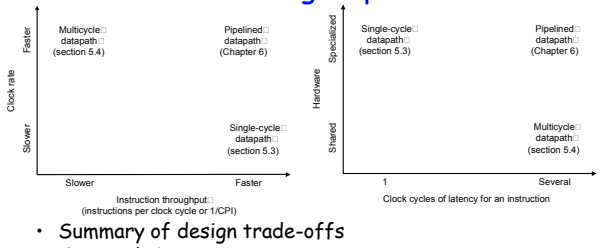
- Delayed branch
  - Instruction after branch always executes
  - Find an independent instruction from before the branch
  - Find instructions from Taken (target) OR from Not Taken (fall-through) code section
- * For Architects



Big picture

- Iron law: Insts/prog * CPI * cycletime
- With pipelining:
  - CPI ~ 1 (with ideal memory, good branch prediction and few data hazards)
  - Cycletime : determined by critical path of one stage

The Design Space

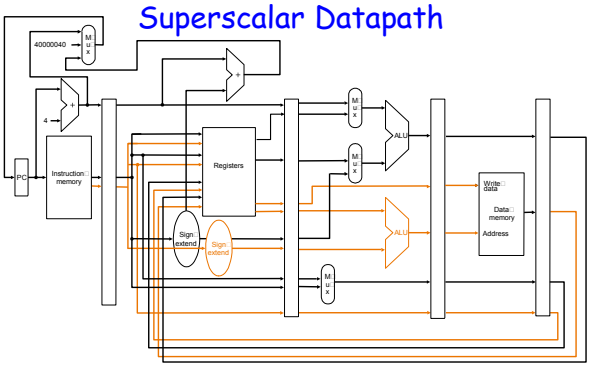


- Summary of design trade-offs
- Can we do better?
  - CPI < 1?
  - i.e., IPC > 1?

Superscalar Processor

- What does it mean?
  - Scalar processors (operate on scalar quantities)
  - Vector (operate on vectors)
  - Superscalar: multiple scalar operations in one cycle
  - More than one instruction per cycle

Superscalar Datapath



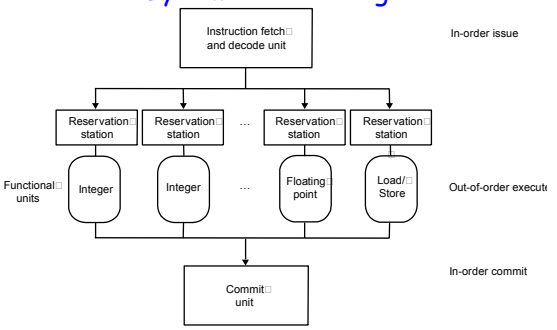
- Replicate datapath elements

Dynamic Scheduling

- No need to suffer hazards if other useful work can be achieved
- Load Hazard results in pipeline stall
  - But other instructions are ready
  - "Oh! But we cannot execute instructions out of order" – Not really

```
lw    $t0, 20($s2)
addu  $t1, $t0, $t2
sub   $s4, $s4, $t3
slti  $t5, $s4, $t3
```

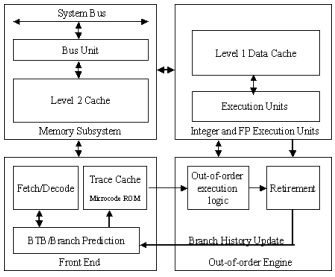

Dynamic Scheduling



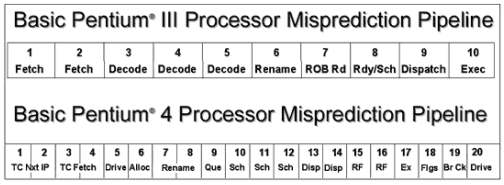
- Instructions can execute when operands are ready
- Instructions can "commit" when all preceding instructions have committed

Pentium 4 on 0.18 micron

- 42 million transistors
- 3GHz
- Several parts are clocked at half the speed
- Inorder front-end, out-of-order execution, in order retire



Pentium 4 pipeline



- Pipeline too much; c.f., Core2