ECE437: Introduction to Digital Computer Design

Chapter 4b (Pipelining)

Fall 2016

Outline

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

Motivation

- We want to improve performance
  - Single-cycle CPU's CPI = 1 but long cycle time
  - Can we shrink cycle time without worsening CPI?
    - Simply 4x faster clock would mean CPI = 4 → no improvement in performance!
    - Breaking up each instr into many cycles is one (only?) way to get faster clock assuming single-cycle CPU implementation is as fast as it can be
  - Break every instr into 5 cycles for fast cycle time and then overlap 5 instrs to make the CPI = 1!
    - Remember CPI is EFFECTIVE CPI
    - We overlap through "pipelining"

RECALL: Iron law

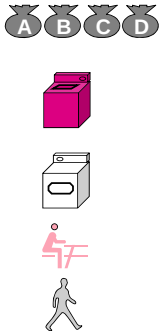
- Time/program = instrs/program x cycles/instr x sec/cycle
  - NEVER forget this!
- sec/cycle (a.k.a. cycle time, clock time)
  - mostly determined by technology and CPU orgn.
- cycles/instr (a.k.a. CPI)
  - mostly determined by ISA and CPU organization
  - EFFECTIVE cycles/instr and NOT actual latency
  - overlap among instructions makes this smaller
    - Each instr 5 cycles but 5 instrs overlap → CPI = 1
  - AVERAGE over instrs (instrs have different CPI)
- instr/program (a.k.a. instruction count)
  - instrs executed, NOT static code
  - mostly determined by program, compiler, ISA

Pipelining

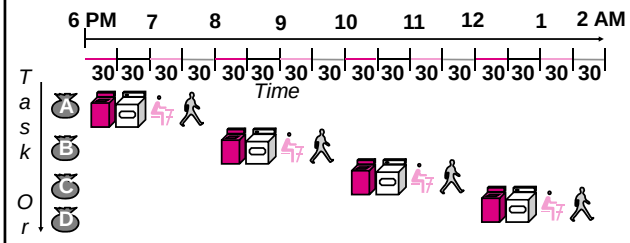
- CENTRAL computer science/engineering concept in BOTH hardware and software
  - There is almost no hardware or software that does not use pipelining for performance!

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

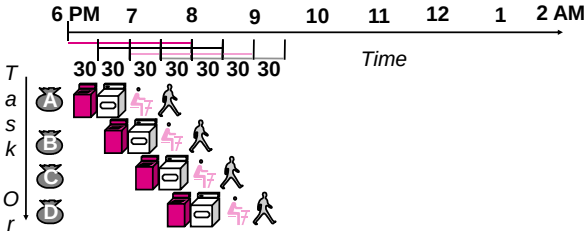


Sequential Laundry



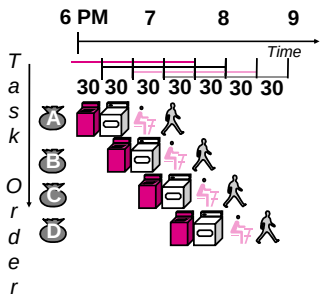
- Sequential laundry takes 8 hours for 4 loads
- If they learned pipelining, how long would laundry take?

Pipelined Laundry



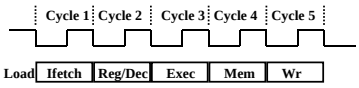
- Pipelined laundry takes 3.5 hours for 4 loads!
- Not esoteric computer architecture concept
- Natural, Intuitive

Pipelining lessons



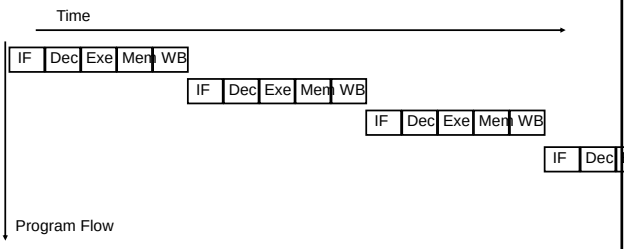
- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipe steps/ stages
- Pipeline rate limited by slowest pipeline stage
- Unbalanced lengths of pipe stages reduces speedup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

The Steps/Stages of Load



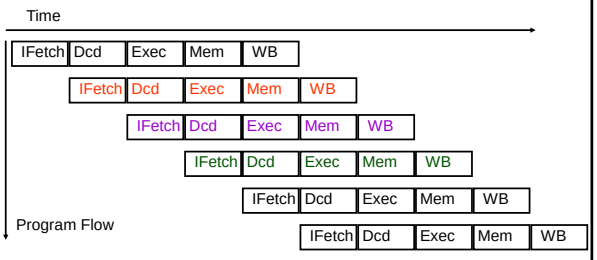
- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec: compute/calculate memory address
- Mem: Read the data from the Data Memory
- Wr: Write the data back to the register file
- REMEMBER THESE 5 STEPS FOREVER!

Single-cycle execution



- When an instruction is in one stage what are the other stages doing?

Pipelined Execution Representation

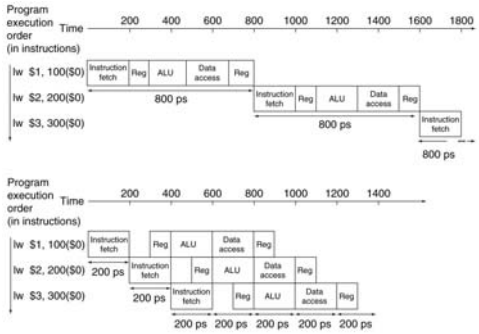


- What is the CPI here?
- Ideal speedup =?

Ideal speedup

- All instructions are executed in 1 cycle in a single-cycle datapath (i.e. CPI = 1)
- Single-cycle cycletime = t ns (say)
- Instr. Count = n
- For P pipeline stages, pipelined cycletime = t/P
- Old time (single-cycle) = n x 1 x t
- New time (pipelined) = n x 1 x t/P + 2 x (P-1) x t/P (fill + drain)
- Speedup = P/(1 + 2(P-1)/n)
- P is some constant, n is large → Speedup = P (= the number of pipeline stages)

Non-uniform pipeline stages

- 

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

Non-uniform stages

Maximum Speedup ≤ Number of stages

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

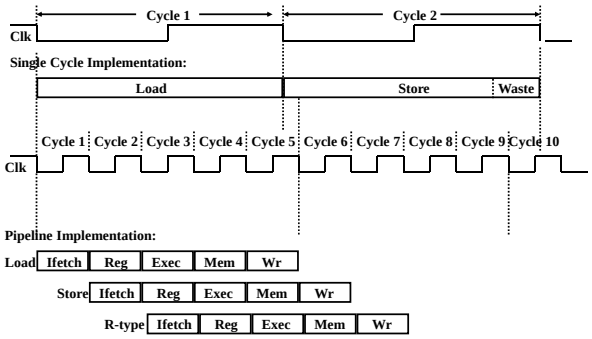
UNLESS you go to asynchronous or self-timed pipelines

- Idea has been around for decades
- Not caught on because very hard to do

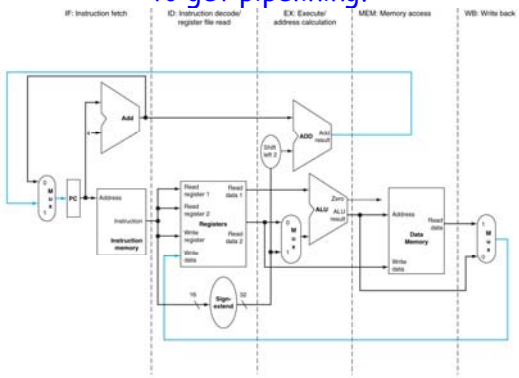
Exercise

- A single cycle processor implementation can be pipelined in two ways
- Pipeline A uses a 5-stage pipeline
  - the 5 stages account for 15%, 10%, 15%, 20%, 40% of the delays respectively
- Pipeline B uses a 3-stage pipeline
  - the stages are balanced
- If instructions are all independent, which pipeline implementation is the better option

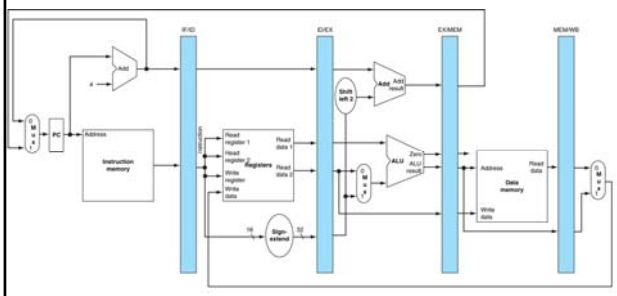
Single Cycle vs. Pipeline



What hardware is added to Single-cycle to get pipelining?



Pipelined Datapath

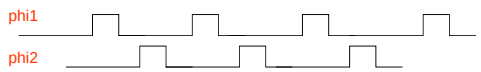


- Pipeline datapath with latches

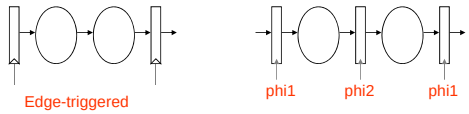
Hardware requirements

- Similar to single cycle pipeline
  - Latches** to separate stages
    - Real machines use latches and multiphase clocks
    - "Latches" == flipflops in this course
- Control
  - Good news
    - Minor changes from single-cycle version

More on Pipeline "latches"



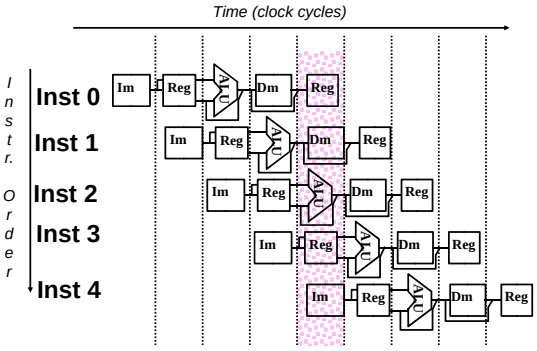
- Remember construction of flip flop
  - Two latches, clock and inverted clock.
- 2-phase non-overlapping clocks
- 1 pipe stage uses two (level-sensitive) latches



Ideal speedup with latches?

- Suppose we execute N instructions
- Single Cycle Machine
  - 45 ns/cycle x 1 CPI x N inst = 45N ns
- Ideal pipelined machine
  - 5 stages but assume slightly slower clock at 10ns and not 9ns (due to latch overhead)
  - 10 ns/cycle x (1 CPI x N inst + 4 cycle drain) = 10N + 40 ns = ~10N for large N
    - Speedup = 4.5

Resources other than latches are already there!



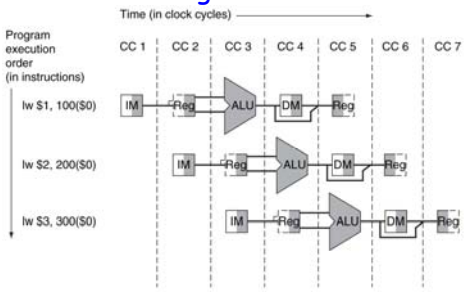
One important thing about latency vs. throughput

- Pipelining does a strange thing
- To improve program latency, pipelining improves instruction throughput without improving instruction latency
  - May actually worsen instruction latency
  - E.g., Pipeline Latch overheads

Pipelined Processor Design

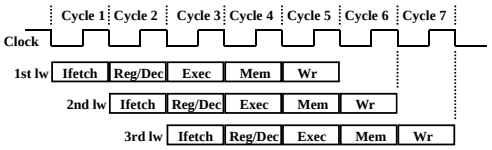
- Designing a pipelined processor
  - Associate resources with states
  - Resources not necessarily atomic or full-cycle
    - Register reads and writes can happen in the same cycle
    - Writes in first half of cycle and reads in the second half – we will see later
  - Assert appropriate controls in each stage
    - Make sure all necessary information is carried through the pipeline along with the instruction

Shading convention



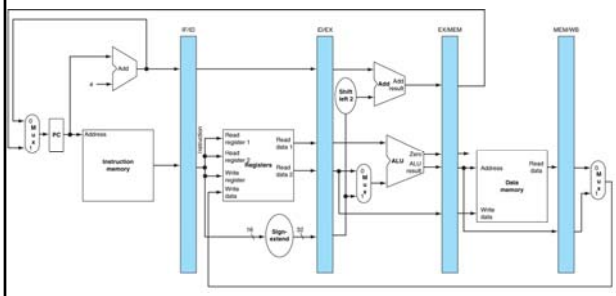
- Shade first half => write shade second half => read
- ALU full-cycle
- Register file can handle read/write in the same cycle
- Memory can handle only 1 read or 1 write per cycle

Pipelining the Load Instruction



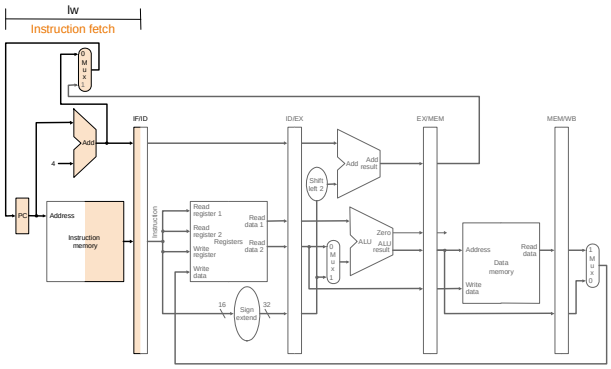
- The five independent blocks in the pipeline datapath are:
  - Instruction Memory for the Ifetch stage
  - Register File's Read ports (bus A and busB) for the Reg/Dec stage
  - ALU for the Exec stage
  - Data Memory for the Mem stage
  - Register File's Write port (bus W) for the Wr stage

Pipelined Datapath

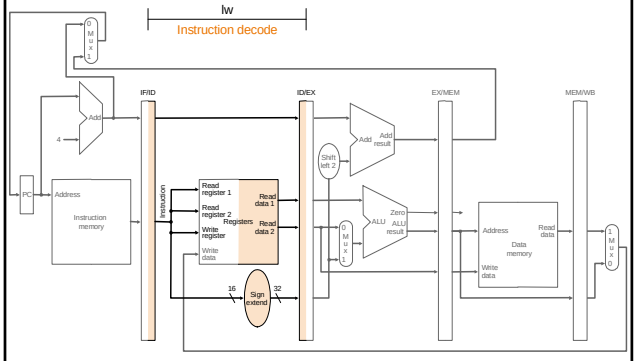


- Pipeline datapath with latches

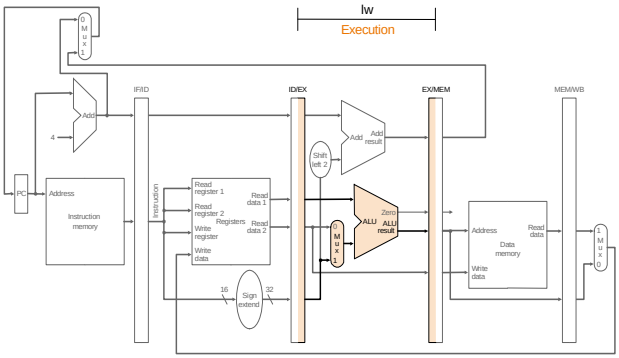
Instruction Fetch



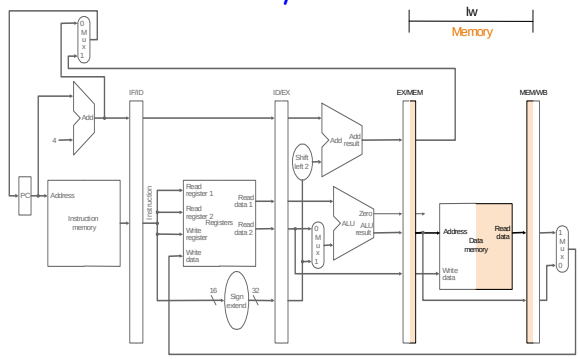
Instruction Decode/Reg Read



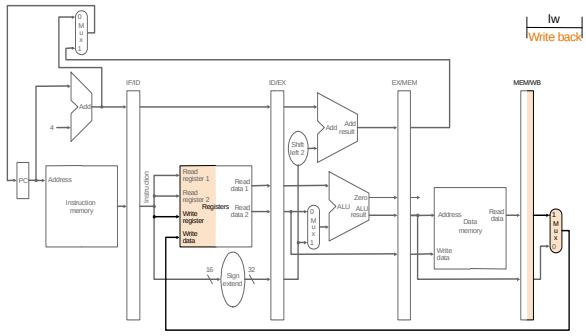
Execute (Address calculation)



Memory Access



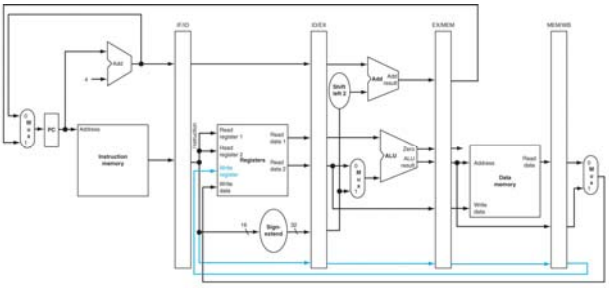
Writeback



Pipeline latches

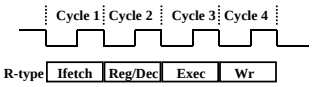
- Length of Pipeline latches
- Book says (Fig 4.35)
  - IF/ID : IR (32), PC+4 (32) : 64 bits
  - ID/EX: IR (32), PC+4 (32) + RegA + RegB : 128 bits
  - EX/MEM: ALUout(32) + Equal(1) + PC+4+SX(imm) (32) : 97
  - MEM/WB: ALUout (32) + MemData(32) : 64
- Inaccurate, will be refined:
  - (see next slide)
  - Other control bits (IR not going through)

Corrected Datapath for lw



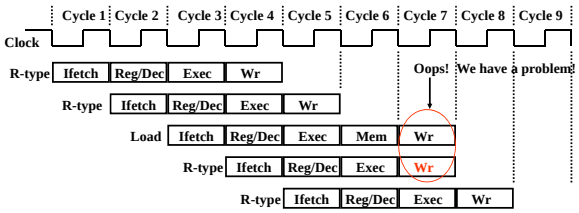
- Carry destination register through pipeline registers to WB stage

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

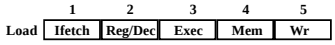
Pipelining R-type and Loads



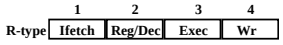
- We have pipeline conflict or "structural hazard":
  - Two instructions try to write to the register file at the same time!
  - Only one write port

Key observation

- Each hardware block can only be used once per instruction
- Each hardware block must be used at the same stage for all instructions:
  - Load uses Register File's Write Port during its 5th stage

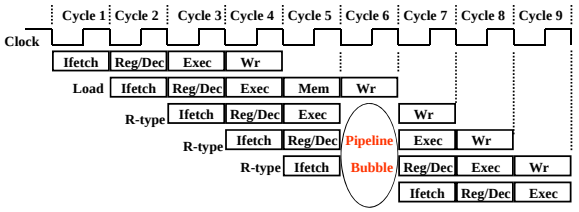


  - R-type uses Register File's Write Port during its 4th stage



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

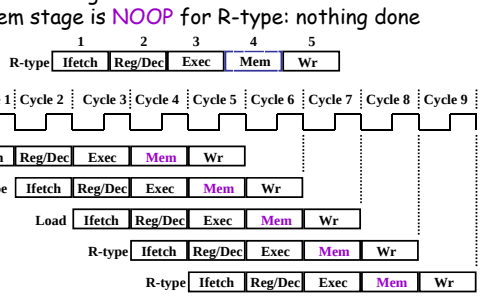
Soln.1: Insert "Bubble"



- 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 – CPI increases!

Soln.2: Delay R-type's Write

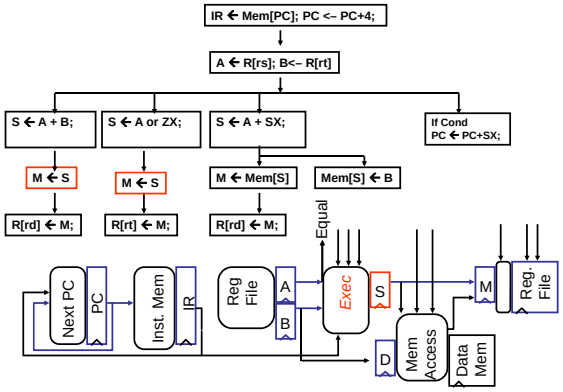
- Delay R-type's register write by one cycle:
  - Now R-type instructions also use Reg File's write port at Stage 5
  - Mem stage is NOOP for R-type: nothing done



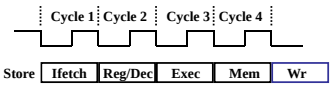
Does soln. 2 lose performance?

- Does it increase R-type's latency?
- Does it worsen R-type's CPI?
- What is the bottomline performance concern in pipelining?

Modified RTL & Datapath

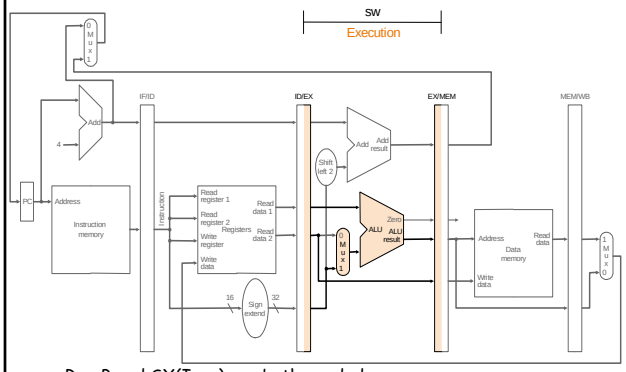


Four Stages of Store



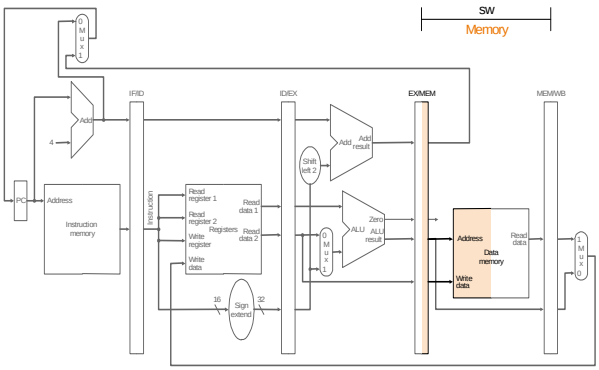
- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec: Calculate the memory address
- Mem: Write the data into the Data Memory

Exec Stage of Store

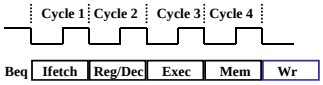


- Reg B and SX(Imm) are both needed

Mem stage of Store



FourStages of Beq



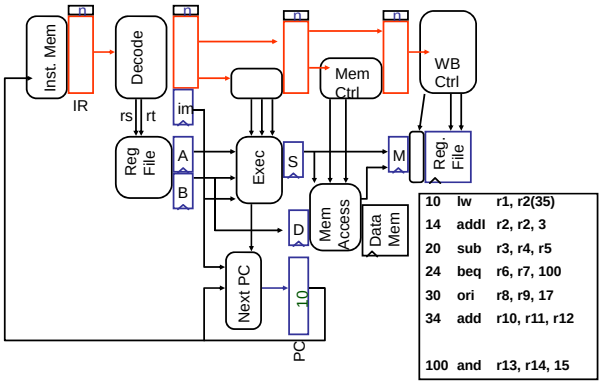
- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- Reg/Dec:
  - Registers Fetch and Instruction Decode
- Exec:
  - compares the two register operand,
  - Compute branch target
- Mem
  - select correct branch target address
  - latch into PC
- There is more to this .....later

Visualizing the pipeline

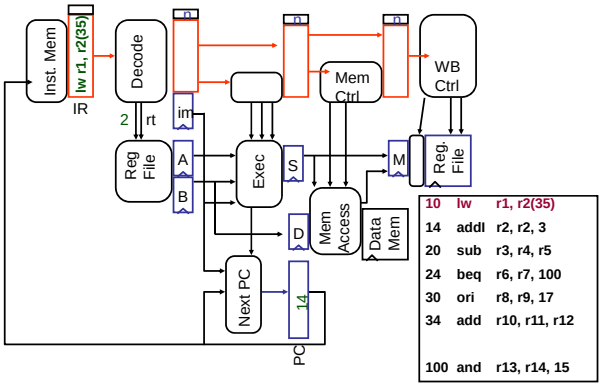
10	lw	r1, r2(35)
14	addI	r2, r2, 3
20	sub	r3, r4, r5
24	beq	r6, r7, 100
30	ori	r8, r9, 17
34	add	r10, r11, r12
100	and	r13, r14, 15

these addresses are OCTAL

Start: Fetch 10

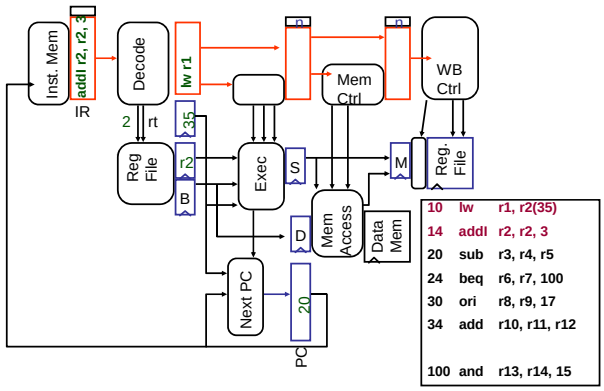


Fetch 14, Decode 10



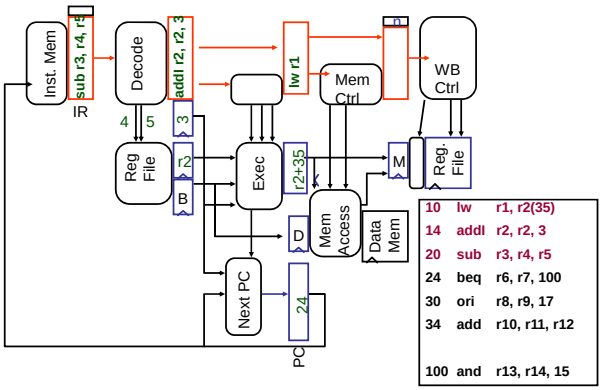
10	lw	r1, r2(35)
14	addI	r2, r2, 3
20	sub	r3, r4, r5
24	beq	r6, r7, 100
30	ori	r8, r9, 17
34	add	r10, r11, r12
100	and	r13, r14, 15

Fetch 20, Decode 14, Exec 10



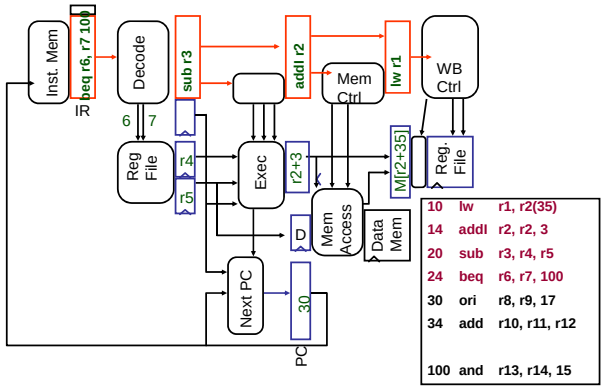
10	lw	r1, r2(35)
14	addI	r2, r2, 3
20	sub	r3, r4, r5
24	beq	r6, r7, 100
30	ori	r8, r9, 17
34	add	r10, r11, r12
100	and	r13, r14, 15

Fetch 24, Decode 20, Exec 14, Mem 10



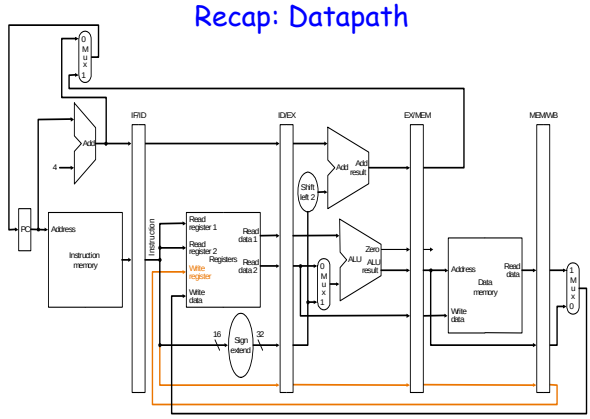
10	lw	r1, r2(35)
14	addI	r2, r2, 3
20	sub	r3, r4, r5
24	beq	r6, r7, 100
30	ori	r8, r9, 17
34	add	r10, r11, r12
100	and	r13, r14, 15

Fetch 30, Dcd 24, Ex 20, Mem 14, WB 10 --- deal with branch later



10	lw	r1, r2(35)
14	addI	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

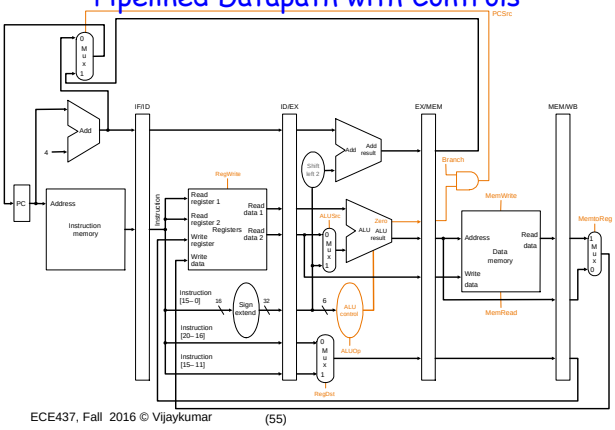
Recap: Datapath



Outline

- Pipeline Datapath
  - Under simplifying assumptions
    - All independent instructions
    - Dependencies – later
      - Datapath not yet capable of handling dependencies
    - Walk-through
- Control
  - Most complicated (read as 'irregular/unstructured')
  - Not that bad for pipelined processors
    - Similar to single cycle control

Pipelined Datapath with Controls



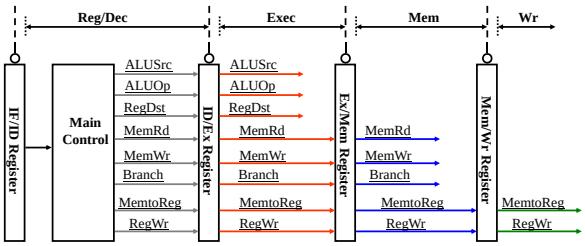
Pipeline Control vs. Single cycle control

- Similar (generation)
- What about
  - PCWrite? IR-write?
  - Write enable for the pipeline registers?
- Pipelined processor implementation
  - Stages common to all instructions
    - Instruction fetch stage (IF)
    - Decode and Register read (ID)
  - Instruction-specific stages
    - Execute (Exec)
    - Memory access (Mem)
    - Write back (WB)

Generating controls

- Simplify the problem
  - Reduce 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

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

Meaning of controls

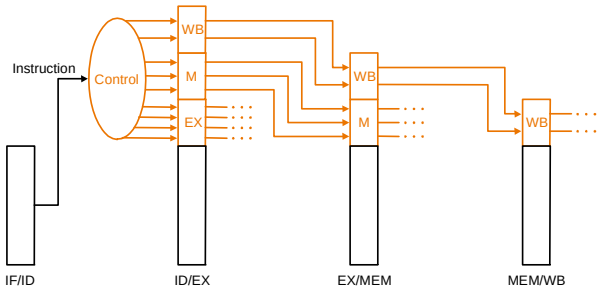
- RegWr : 1-> write, 0-> no write
- MemToReg : 1-> MDR, 0-> ALUOut
- RegDst : 1->rd, 0->rt
- ALUOp<1:0> : 00->Add,01->Sub,10->'funct'
- ALUSrc : 0->RegB, 1->SX(Imm)
- Branch: 0->non-branch inst, 1-> branch
- MemRead: 1->memread, 0-> no memread
- MemWrite: 1->memwrite, 0->no memwrite
- ALU control abstracted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

WB <1:0>

Exec <3:0>

Mem <2:0>

Extended Pipeline Registers



- Simple extensions to carry control state

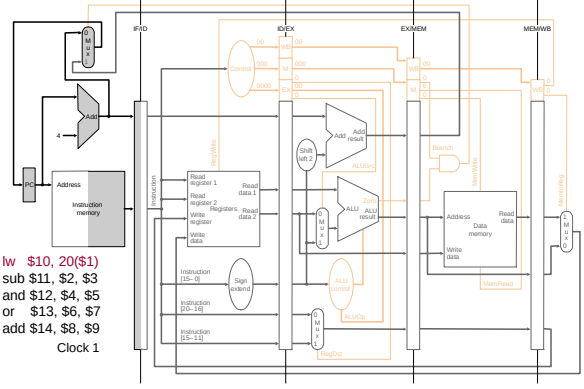
Pipeline Walkthrough with controls

- 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

Pipeline Walkthrough with controls

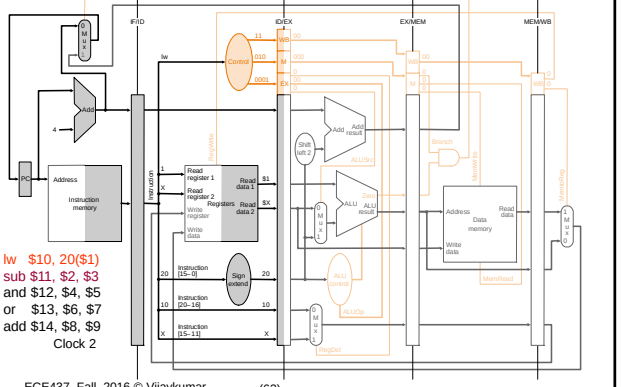
IF: lw $10, 20($1) | ID: before<1> | EX: before<2> | MEM: before<3> | WB: before<4>



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

Pipeline Walkthrough with controls

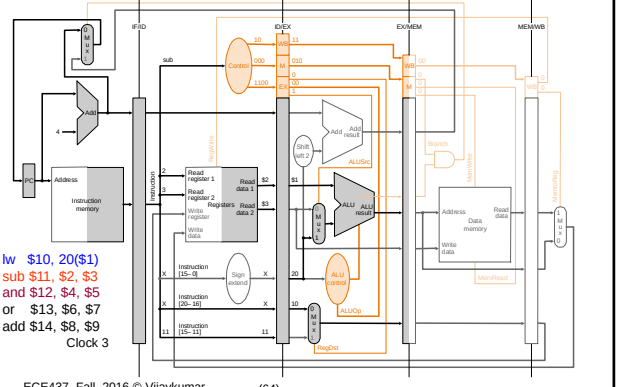
IF: sub $11, $2, $3 | ID: lw $10, 20($1) | EX: before<1> | MEM: before<2> | WB: before<3>



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

Pipeline Walkthrough with controls

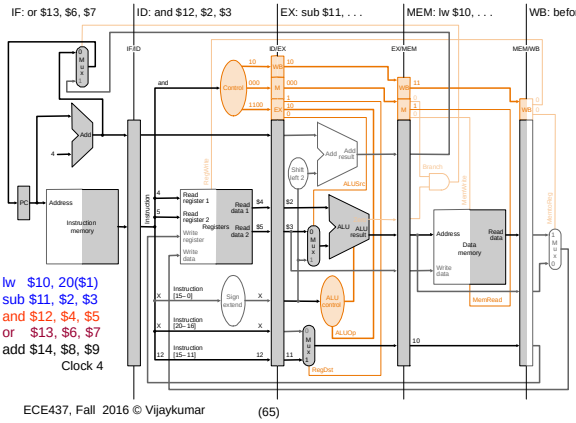
IF: and $12, $4, $5 | ID: sub $11, $2, $3 | EX: lw $10, . . . | MEM: before<1> | WB: before<2>



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

Pipeline Walkthrough with controls

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

Pipeline Walkthrough with controls

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

Pipeline Walkthrough with controls

IF: after<1> | ID: add $14, $8, $9 | EX: or $13, . . . | MEM: and $12, . . . | WB: sub $11,

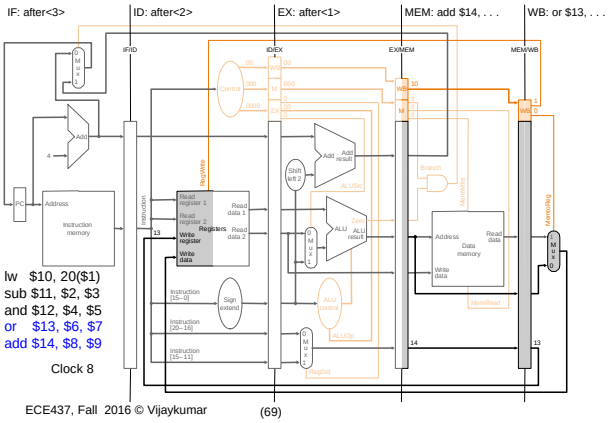
lw $10, 20($1)
sub $11, $2, $3
and $12, $4, $5
or $13, $6, $7
add $14, $8, $9
Clock 6

Pipeline Walkthrough with controls

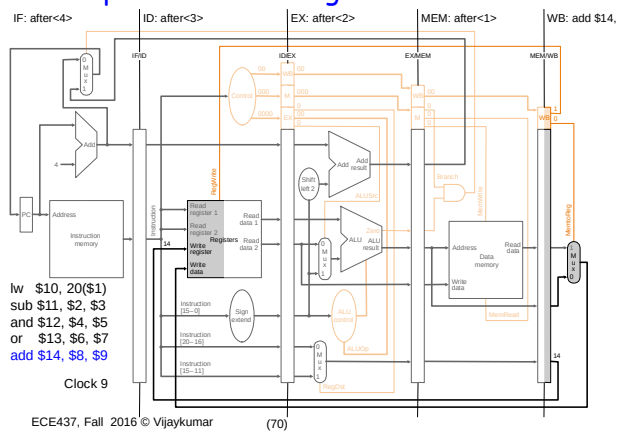
IF: after<2> | ID: after<1> | EX: add $14, . . . | MEM: or $13, . . . | WB: and $12

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

Pipeline Walkthrough with controls



Pipeline Walkthrough with controls



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
- Keep memory controller (arbiter) SEPARATE from pipeline (as you did for single-cycle)
  - In IF and MEM, send request to arbiter

Summary

- Only slightly more complicated than single cycle
  - not really, seemingly so because we:
    - ignored complications of DEPENDENCIES
  - Need deeper understanding of dependencies
    - need to modify datapath as well

Hard part of pipelining

- Definitely:
  - MUST 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
  - data hazards: attempt to use item before it is ready
    - instruction depends on result of prior instruction still in the pipeline
  - control flow hazards: attempt to make a decision before condition is evaulated
    - branch instructions

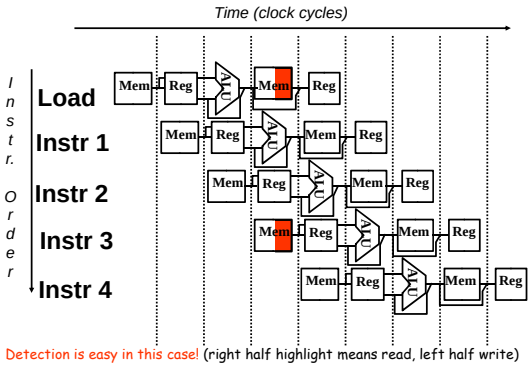
Hazards

- Structural hazards
  - Two instructions need the same hardware at the same time
- Data Hazards
  - Data not ready
- Control flow 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
  - Increase CPI
  - Reduce speedup

Single Memory: Structural Hazard



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

Structural Hazards

- Single memory (suppose)
- 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 at least 1.3 (otherwise memory is used more than 100%)
- Solution?

Data Hazards

add r1 ,r2,r3

sub r4, r1 ,r3

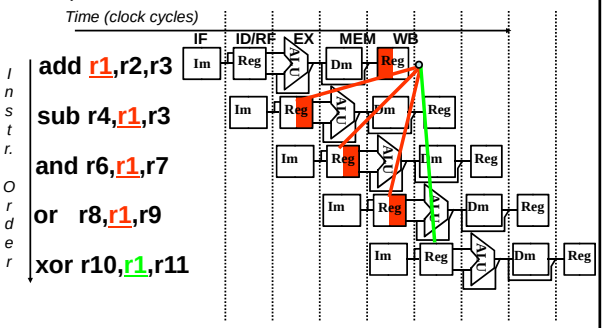
and r6, r1 ,r7

or r8, r1 ,r9

xor r10, r1 ,r11

Hazards on r1

- Dependencies backwards in time



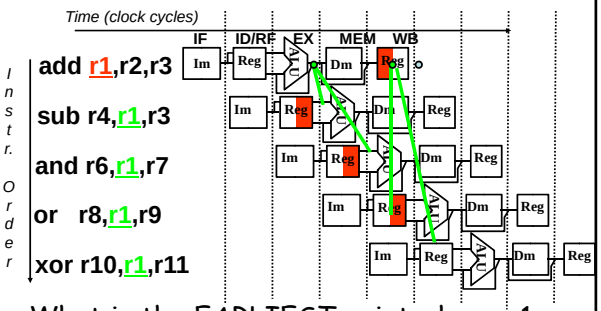
Data Hazard Solution1: Stall

- Can always stall until hazard goes away
  - Delay sub and later instrs till add is in WB
  - Increase CPI – lose performance
- But performance loss bad only if common case
  - Amdahl's law
- So what about data hazards?
  - Think about how code looks

Data Hazard Solution1: Stall

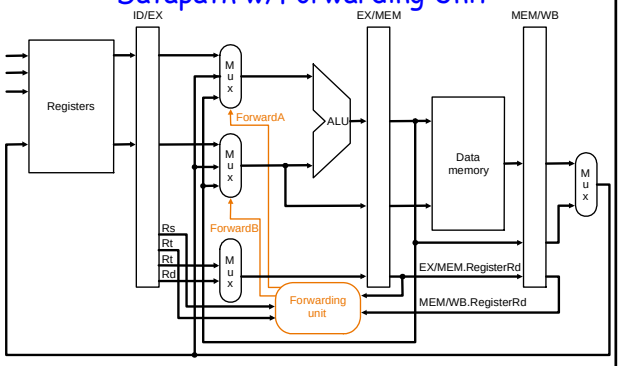
- So what about data hazards?
  - Think about how code looks
    - Key difference between circuit designer and computer architect
    - circuit people do not think about code
    - Architects think about interaction of code with hardware – common case interactions
  - VERY COMMON code:
    - x = a + b;
    - = use x; /* IMMEDIATELY use x */

Data Hazard Solution2: forwarding (a.k.a. bypassing)



- What is the EARLIEST point where r1 is available SOMEWHERE in the pipe

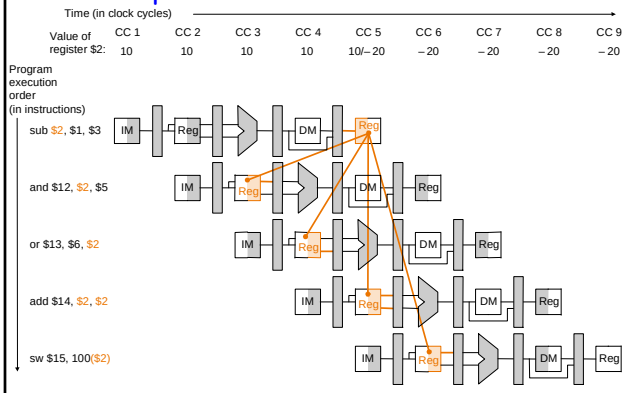
Datapath w/Forwarding Unit



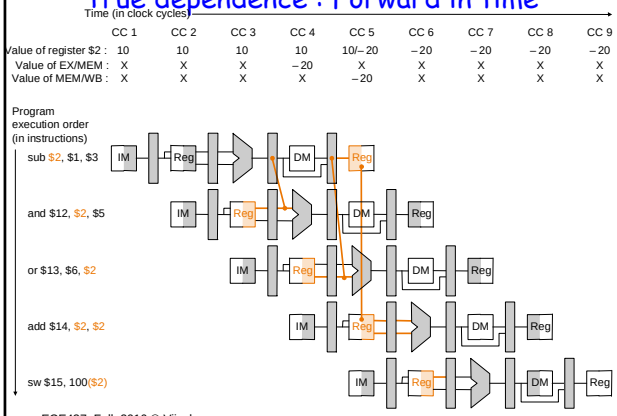
b. With forwarding

- Forwarded values from pipeline latches

Dependence : Backward in time

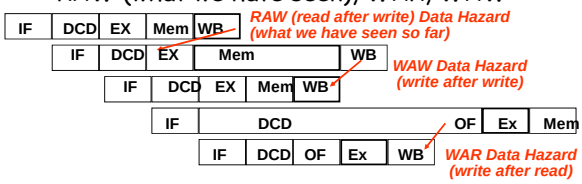


True dependence : Forward in time

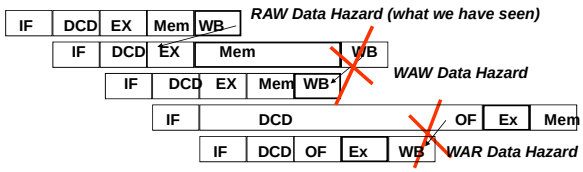


Other types of Data Hazards

- Challenge: maintain illusion of sequential execution
- Types of data hazards
  - RAW (what we have seen), WAR, WAW



Data Hazards



- Avoid some "by design" (as done in MIPS pipeline)
  - eliminate WAR by always fetching operands early (DCD) in pipe
  - eliminate WAW by doing all WBs in last stage
- Detect and resolve remaining ones (RAW)
  - stall or forward (if possible)

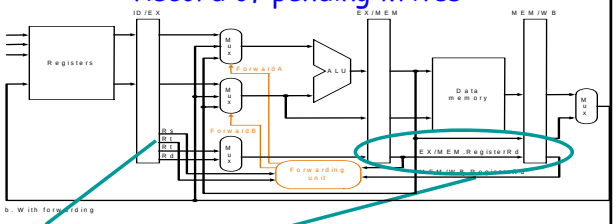
Handling RAW Hazards

- 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
  - Some instructions may not write to register file (store, branch)

Detecting RAW hazards

- Suppose instruction i is in EX and a predecessor instruction j is in a later stage
- A RAW hazard may exist on register ρ if $\rho \in$ Rregs(i) $\cap$ Wregs(j)
  - Compare pending writes (for inst's in later stages) with operand regs of current instruction.
- A WAW hazard may exist on register ρ if $\rho \in$ Wregs(i) $\cap$ Wregs(j)
- A WAR hazard may exist on register ρ if $\rho \in$ Wregs(i) $\cap$ Rregs(j)
- MIPS: RAW hazards only

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

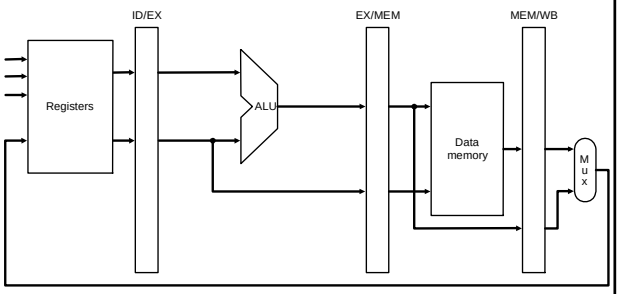
- Restatement of equations
- Text book version
  - WB stage is not really a hazard
    - written in first half of cycle, read in 2nd half
  - EX/MEM.RegisterRd == ID/EX.RegisterRs
  - EX/MEM.RegisterRd == ID/EX.RegisterRt
  - MEM/WB.RegisterRd == ID/EX.RegisterRs
  - MEM/WB.RegisterRd == ID/EX.RegisterRt

Lookahead: Forwarding datapath

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

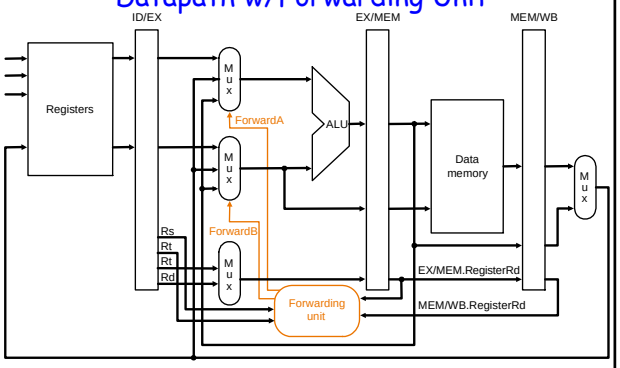
Base Pipelined Datapath

- Simplified representation of pipelined datapath
  - To avoid clutter



a. No forwarding

Datapath w/Forwarding Unit



b. With forwarding

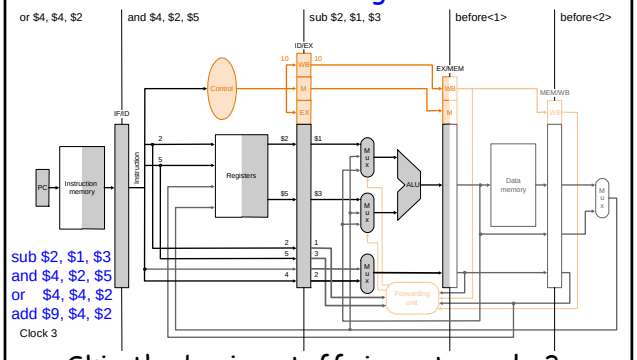
- ForwardA/ForwardB: 01->Mem, 10->EX

Data Hazards and Forwarding: Walkthrough

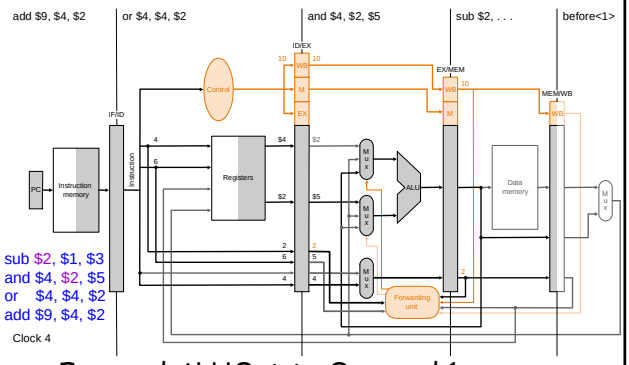
- Code snippet
  - identify hazards
  - identify forwarding paths

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

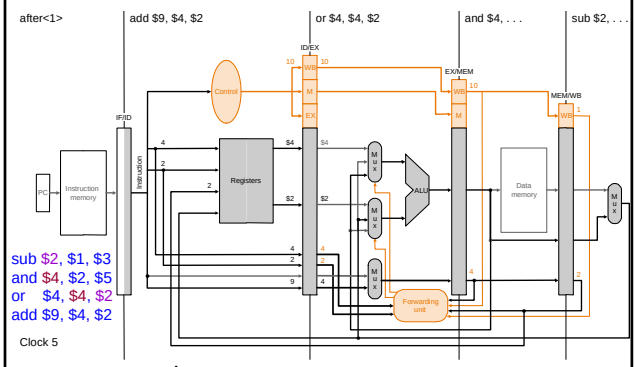
Walkthrough



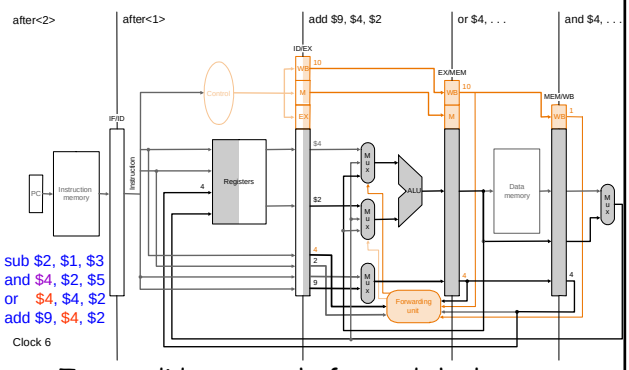
- Skip the boring stuff, jump to cycle 3



- Forward ALUOut to Operand 1

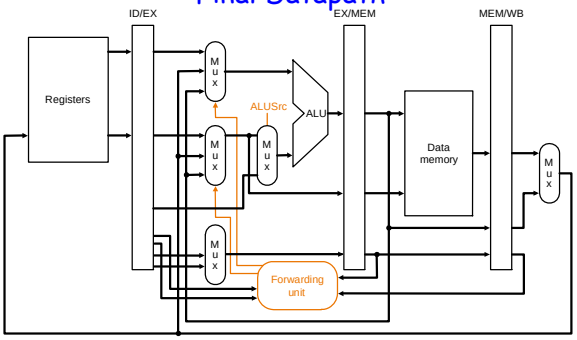


- Forward ALUout to Op1, Mem to Op2



- Two candidates match, forward the latest

Final Datapath



- For "Imm" op, ALUsrc mux (Fig 4.57)

Control for Forwarding

- 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
```


Control for Forwarding

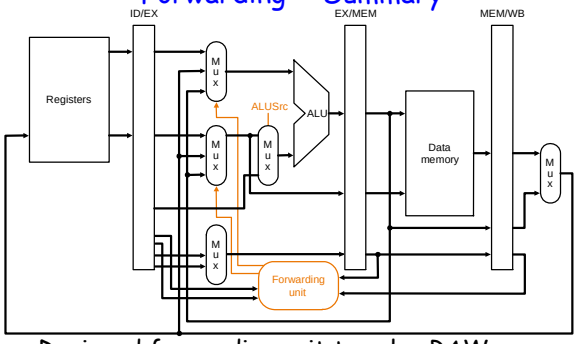
- MEM hazard

```
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 ?

Forwarding - Summary

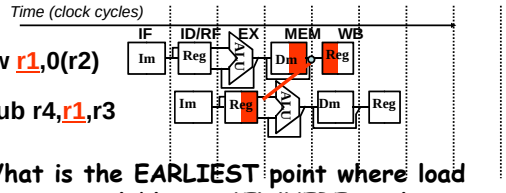


- Designed forwarding unit to solve RAW hazards for R-type instructions

Pay attention

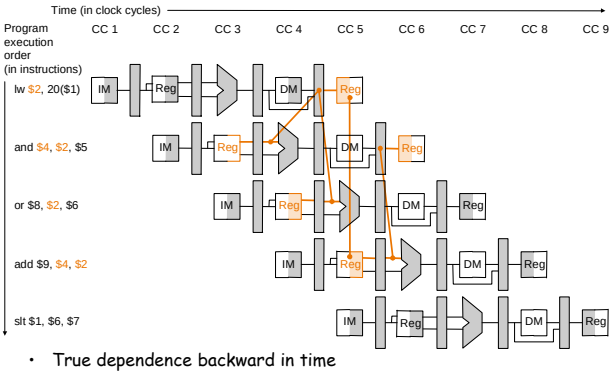
- Forwarding is FROM EX/MEM or MEM/WB pipeline latches TO INSIDE EX (not to latch)
- Other choices
  - eg FROM ALU output TO ID/EX pipeline latch
  - or FROM EX/M or M/W pipeline latches to ID/EX latch
  - will not work or cause stalls or require larger pipeline latches

Forwarding does not solve all RAW hazards



- **What is the EARLIEST point where load data is available SOMEWHERE in the pipe**
  - **KEY difference between R-type and ld**
  - **Value available: for R-type after EX, for ld after MEM**
- **Can't solve with forwarding:**
  - **Must delay/stall instruction dependent on loads**

Load instruction



- True dependence backward in time

Ld-use hazard Solution

- Catch-all solution for hazards
  - Stall if instruction immediately next to a load is dependent on the load
    - always works, but hurts performance
    - Use as last resort
  - Can't help: load value is unavailable, unlike R-type instructions
- Challenge:
  - Modify pipeline to stall for such hazards
  - Detect and stall LATER instructions

Terminology

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

Ld-use Detection

- Conditions
  - IMMEDIATELY preceding instruction must read memory
    - MemRead must be asserted
  - Destination of preceding instruction (rt) must be one of operands of current instruction

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

- 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

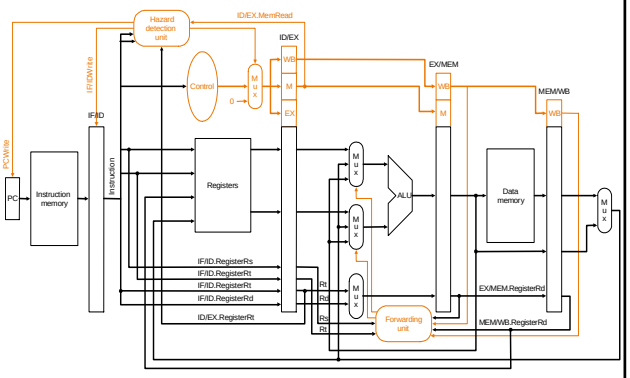
Stalling the pipeline

- Current instruction cannot proceed
  - Later instructions must be stalled too else they will run into each other
- Solution for RAW hazards
  - inject NOP into EX/Mem pipeline
  - Prevent writes to PC and IF/ID register
  - Earlier instructions proceed as usual
- GENERAL solution for all stalling
  - Inject nop instead of stalled instr in next stage
  - "freeze" later instrs behind in previous stages
  - "continue" earlier instrs ahead in later stages

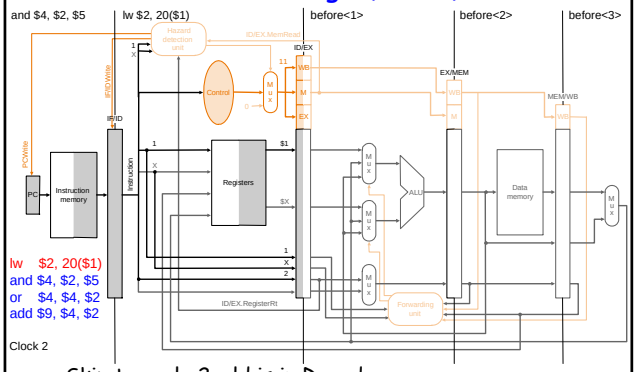
Stalling the pipeline

- GENERAL solution for all stalling
- WHILE stalled
  - inject nop in place of stalled instr in next stage
    - As many nops as stalls
  - "freeze" later instrs behind (in earlier stages)
    - As long as stalled
  - "continue" earlier instrs ahead (in later stages)

Datapath

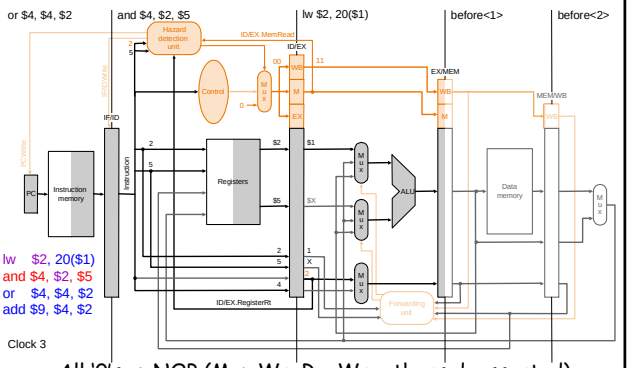


Walk-through (1 of 6)



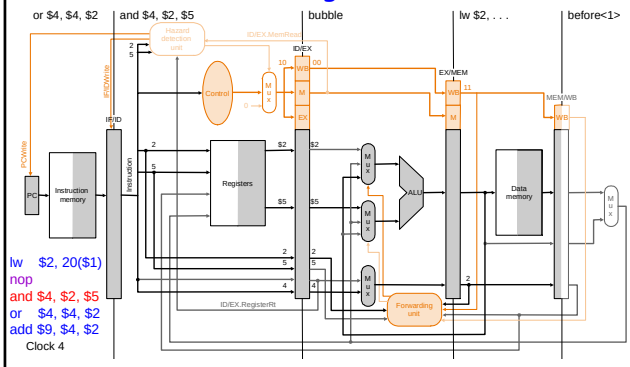
- Skip to cycle 2 – ld is in Decode

Walk-through (2 of 6)

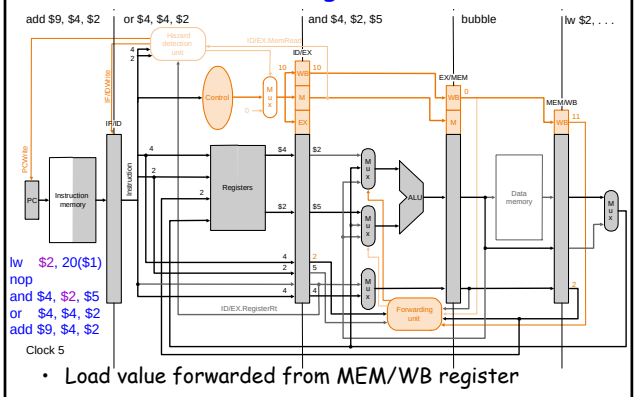


- All '0's => NOP (MemWr, RegWr, others deasserted)

Walk-through (3 of 6)

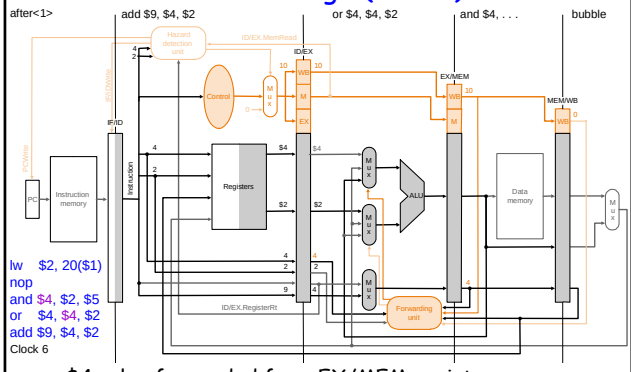


Walk-through (4 of 6)



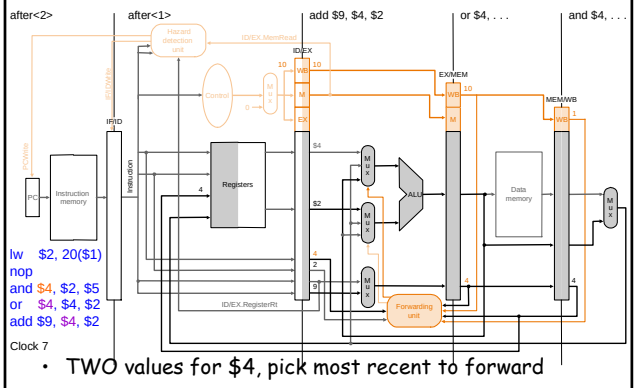
- Load value forwarded from MEM/WB register

Walk-through (5 of 6)



- $4 value forwarded from EX/MEM register

Walk-through (6 of 6)

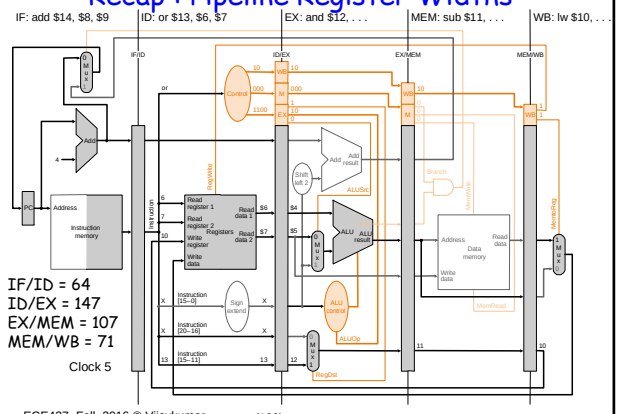


- TWO values for $4, pick most recent to forward

RAW Hazard with Loads: Summary

- True backward dependencies in time
  - Need to stall
  - Dependent instruction & later instructions stalled
  - Earlier instructions can proceed
- Stall achieved by
  - Detecting hazard (remember logic equation)
  - Inserting NOP (all EX/MEM/WB controls set to 0)
  - Preventing IF/ID latch and PC from being overwritten

Recap : Pipeline Register Widths



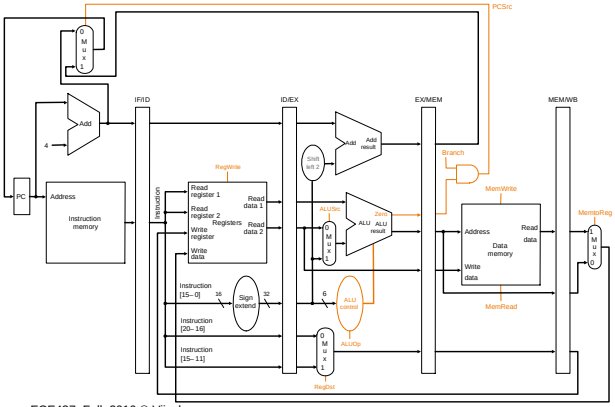
Solution3 for RAW Hazard with Loads

- We can use the compiler to insert nops between ld and use to prevent hazards
- But we can do better
  - Ld $16, 0[$24]
  - Nop (inserted by compiler or hardware)
  - Add, $17, $16, $18
  - Sub $23, $8, $9
  - St $17, 0[$25]
- What can you do?

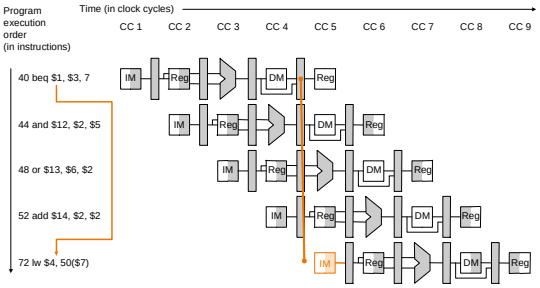
Recall Hazards

- Structural hazards
  - Two instructions need the same hardware
  - Half/half for REG and ch5 for MEM
- Data Hazards
  - Data not ready
  - Forward/bypass (stall for loads)
- Control flow Hazards – next
  - Which instruction to fetch? Not known.
  - Delayed branch, Predict not taken

When are 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

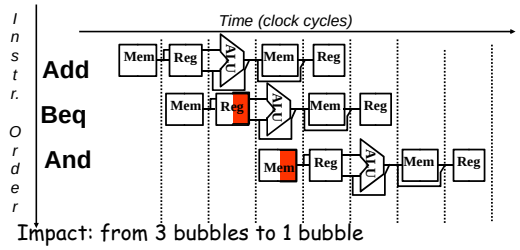
- Branch resolved in the MEM stage
  - Which is 3 cycles later but next instruction has to fetched in the next cycle
- Solution 1: Stall but penalty is 3 bubbles
  - Oh but how common is this?
    - Amdahl's law – worry about it only if common
  - Every 6th instruction is a branch
    - CPI would go from ~1.2 to 1.2+1/6*3 = 1.7 (why 1.2?)
    - Large performance loss
- There are 3 other solutions

Hazards in real microprocessors

- Some of you may be thinking – 1-3 bubbles for loads and branches ain't that bad – performance impact is less than 2x
- But real pipelines are 10-15 stages deep, so without doing anything each load and branch would incur 5-10 bubbles!
  - Well then why are real pipelines so deep?
  - To get fast clock via many schemes (taught in 437) in late 80's thru mid 90's

Control flow Hazard: Solution2

- We can move up branch decision to Decode by adding h/w to compare registers as they are being read

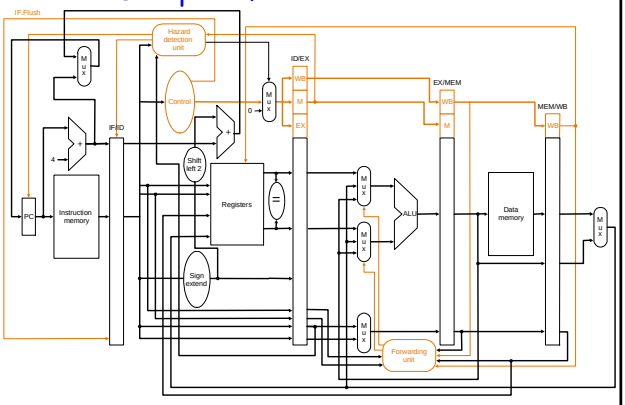


- Impact: from 3 bubbles to 1 bubble

Solution 2

- move branch decision earlier in pipeline
  - Need additional comparator (r1=r2?) and adder (PC+4+SX(IMM)*4)
  - Recall branch needs both outcome & target
  - "flush" IF/ID latch to kill potentially incorrect fetch immediately after branch
  - Will this work well?

Datapath for branch hazards



Br Solution 2

- move branch decision earlier in pipeline
  - Whoa! Value needed in earlier stage
    - what if r1/r2 write is pending?
    - Add r1, r2, r3
    - Beq r1, 0, target // is this common?
    - Forwarding and/or stalling
    - Forwarding from EX/MEM and MEM/WB into decode (previously it was into EX)
      - With forwarding only 1 bubble
      - Are these cases common?
      - Too much! (and better solution exists)

Br Solution 2

- If you use solution2 in the lab then you need to make sure to latch the branch target at the end of decode (it is not latched in previous slide) else br target will be lost if you stall fetch for some (other) reason (e.g., due to memory structural hazard) WHILE a br is in decode

Can we do anything about br stalls?

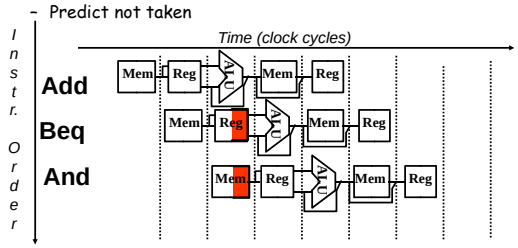
- RAW hazards a problem for both branches and R-types
  - Value produced later but needed earlier
  - R-type: Written in WB and needed in ID
  - Br: Outcome produced in MEM and needed in ID
- But solutions are fundamentally different
  - There is a key difference between branches and R-types – what?

Can we do anything about br stalls?

- Solution 3
  - Predict branch is always not taken
    - MUCH more sophisticated prediction possible
    - Why not predict taken?
  - Soution 4: Delay slots
    - Compiler's problem
- Walkthrough example for solution 3
  - Predict not taken

Control flow Hazard: Solution3

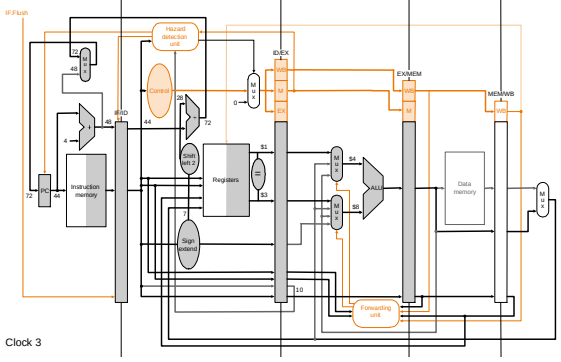
- Predict: guess one direction then back up if wrong
  - Predict not taken



- Impact: 0 bubbles if correct, 1 bubble as before if wrong (correct say 50%)
- More dynamic scheme (correct 90+%)

Walkthrough (1 of 2)

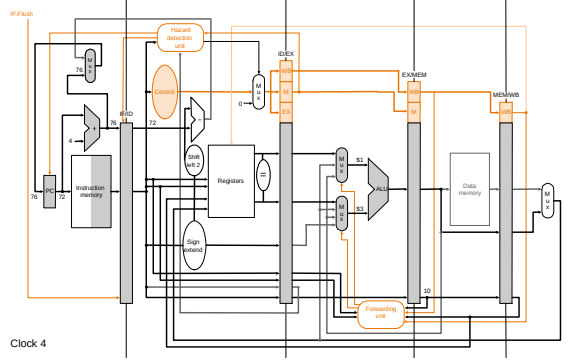
and $12, $2, $5 | beq $1, $3, 7 | sub $10, $4, $8 | before<1> | before<2>



Clock 3

Walkthrough (2 of 2)

lw $4, 50($7) | bubble (nop) | beq $1, $3, 7 | sub $10, . . . | before<1>



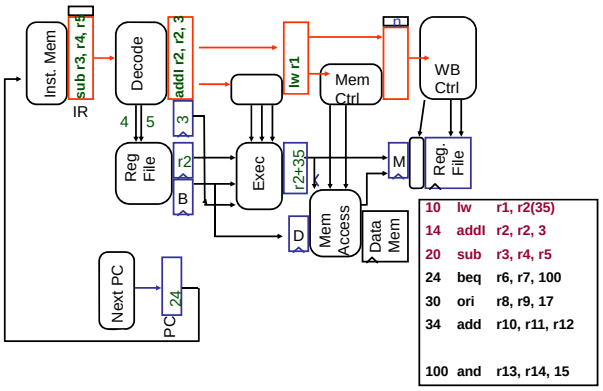
Clock 4

Visualizing solution 3

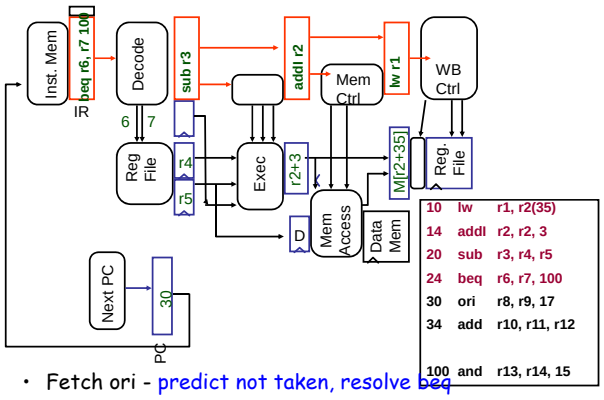
10	**lw**	**r1, r2(35)**
14	**addI**	**r2, r2, 3**
20	**sub**	**r3, r4, r5**
24	**beq**	**r6, r7, 100**
30	**ori**	**r8, r9, 17**
34	**add**	**r10, r11, r12**
100	**and**	**r13, r14, 15**

these addresses are octal

Fetch 24, Decode 20, Exec 14, Mem 10



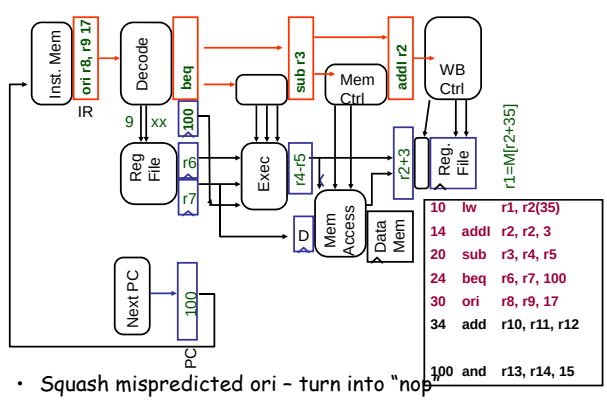
Fetch 30, Dcd 24, Ex 20, Mem 14, WB 10



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

- Fetch ori - predict not taken, resolve beq

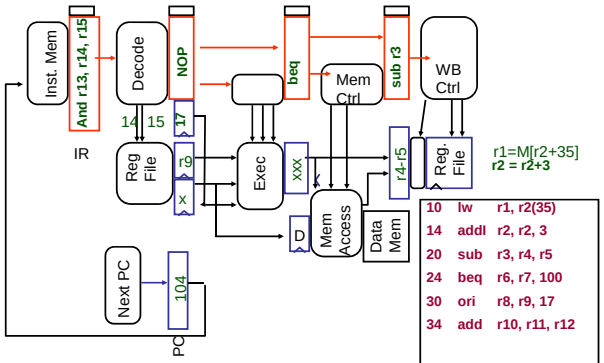
Fetch 100, Dcd 30, Ex 24, Mem 20, WB 14



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

- Squash mispredicted ori – turn into "nop"

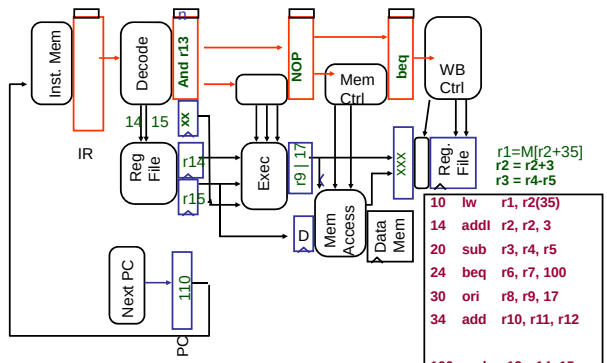
Fetch 104, Dcd 100, Ex 30, Mem 24, WB 20



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

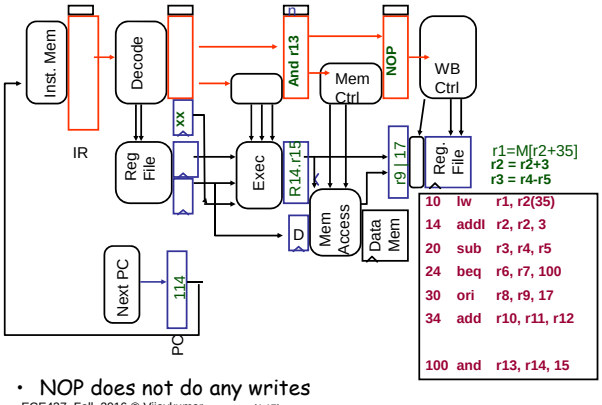
- Nop'd ori flows through pipe

Fetch 110, Dcd 104, Ex 100, Mem 30, WB 24



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

Fetch 114, Dcd 110, Ex 104, Mem 100, WB 30



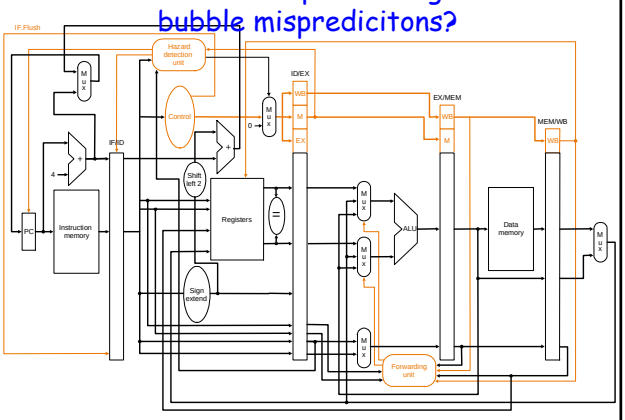
10	lw	r1, r2(35)
14	addI	r2, r2, 3
20	sub	r3, r4, r5
24	beq	r6, r7, 100
30	ori	r8, r9, 17
34	add	r10, r11, r12
100	and	r13, r14, 15

- NOP does not do any writes

Br Solution 3

- If we assume that br decision is NOT moved up (as in solution 2) then
  - Brs cause 3-cycle stall
  - With prediction (solution 3)
    - If correct 0 bubbles and 3 bubbles otherwise (as before)

How will this datapath change for 3-bubble mispredicitons?



Dynamic Branch Prediction – take solution3 further

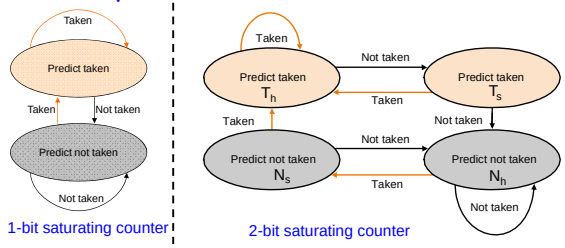
- Better (ie more accurate) than static prediction of not-taken
  - Branches are biased → predictable
  - Some are mostly taken and others mostly not-taken
  - Why? Think of code (architects always think of s/w)
  - Give egs of mostly taken and mostly not-taken

Dynamic Branch Prediction – take solution3 further

- ~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

- Then how does h/w learn the bias of each branch?

Dynamic Branch Prediction



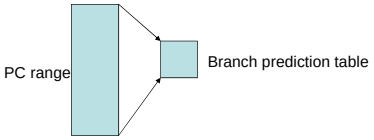
- Store each branch instruction's history ***
  - If a branch is taken in "recent" history, assume the branch is biased to taken and predict taken
    - One bit saturating counter
    - Two bit counters

1-bit vs. 2-bit

- While (always)
  - for i = 0 to 3 ..
    blabla
-
-
- Outcome: TTTNTTTNTTTNTTTN..
- 1-bit (init N): NTTTNTTTNTTTNTTT..
  - 2 mistakes per loop: one each at entry, exit
  - Ignore init mistake
- 2-bit (init N_h): $N_hN_sT_hT_hT_sT_hT_hT_hT_sT_hT_hT_hT_sT_hT_hT_h$..
  - 1 mistake per loop: at exit

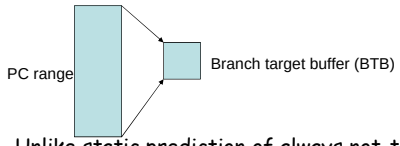
```
top:  mov ri, 0
loop: blabla
      add ri, ri, 1
      blt ri, 3 loop
      j  top
```


Branch Prediction Table



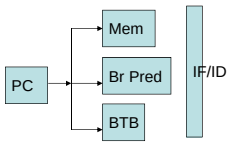
- Store each branch's history ***
  - Not really
- Keep a small table indexed by program counter
- PC is large (32 bit number)
- Many-to-few mapping from PC to table entries
  - E.g. 2048-entry branch prediction table
  - Mapping: use 11 bits of PC
- Problem: Multiple branches may map to same entry in table – Aliasing

Branch Prediction: Branch Target Buffer



- Unlike static prediction of always not-taken, dynamic prediction will predict both taken and not-taken
- For taken prediction, we have a problem (what?)
- There is something about branch targets that helps us solve the problem (what?)
- Keep a small table of br targets indexed by PC (BTB)
  - WHILE fetching (potentially) a branch, use branch PC to look up prediction AND target of branch so NEXT cycle you can fetch taken instruction if predict taken else fetch not-taken instruction - back-to-back fetch for both taken or not-taken
- Many-to-few mapping from PC to BTB

Fetch with branch prediction



10	lw	r1, r2(35)
14	addI	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

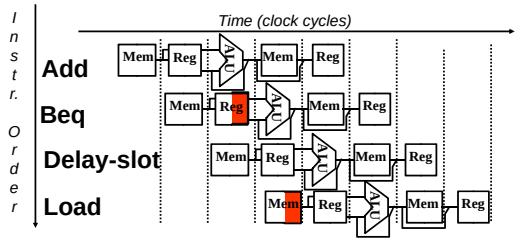
- WHILE fetching (potentially) a branch, use branch PC to look up prediction AND target of branch so NEXT cycle you can fetch taken instruction if predict taken else fetch not-taken instruction - back-to-back fetch for both taken or not-taken
- beq F | D | X | M | W
- and F | D | X ← no bubble

Recap

- Branch instructions
  - Control flow hazard
  - Static branch prediction
    - Predict not taken
    - Squash instruction if prediction incorrect
- Dynamic Branch prediction
  - 1-bit and 2-bit state machines to track history of branches
  - Finite table
    - Potential for "aliasing"
    - Multiple branches map to the same predictor

Control flow Hazard: Solution4

- Redefine branch behavior to take effect after next instruction) "delayed branch"



- Impact: 0 bubbles per branch if can find instruction to put in "delay slot" (say 60% of time)
- As pipelines get deeper, less useful – why?

Delayed Branch: Solution 4

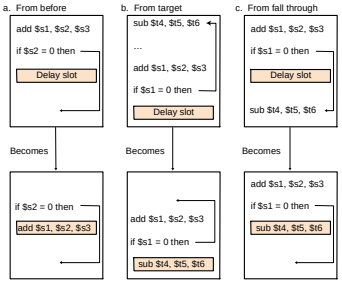
10	lw	r1, r2(35)
14	addI	r2, r2, 3
20	sub	r3, r4, r5
24	ori	r8, r9, 17
30	beq	r6, r7, 100
34	NOP	
38	add	r10, r11, r12
100	and	r13, r14, 15

10	lw	r1, r2(35)
14	addI	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

- Delayed branch: inst after branch ALWAYS executed
  - Branch effect is delayed by 1 cycle – hence the name
  - Invisible to programmer
  - Compiler and/or assembler transforms code

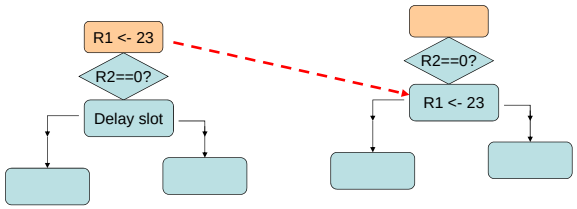
Easy way*** to hide branch hazard delay

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



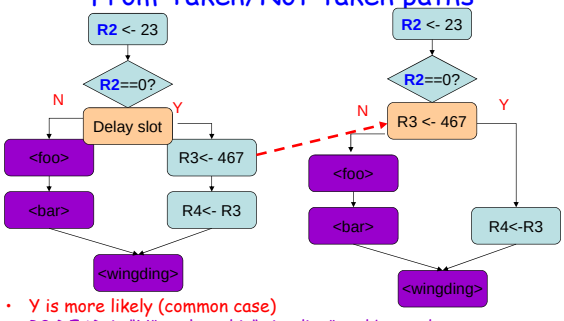
- *** For Architects

Ideal – from before branch



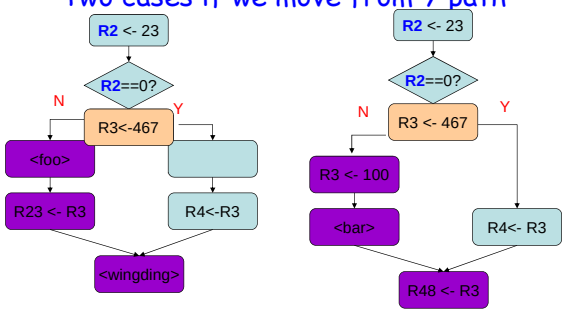
- Independent instructions to fill delay slot
- Code transformation must preserve original semantics
  - Correctness
- But may not find independent instruction from before branch then try from taken or not-taken path

From Taken/Not-taken paths



- Y is more likely (common case)
- R3 DEAD in "N" path and in"wingding" and beyond
  - DEAD = overwritten before being read (KEY correctness requirement)
- Delay slot
  - Profitable mostly, not profitable occasionally, but NEVER unsafe (incorrect)

Two cases if we move from Y path



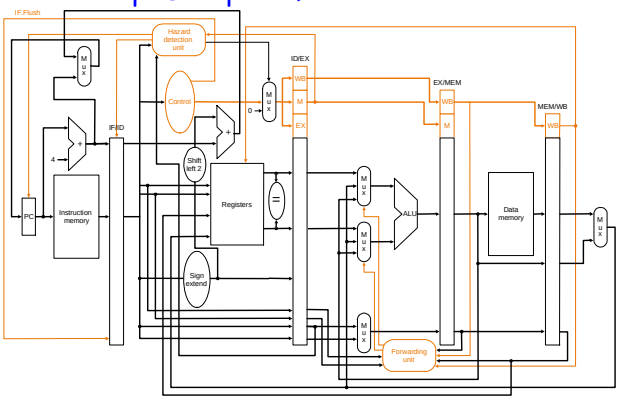
- NOT OK (R3 alive in N path)

OK (R3 dead in N path)

Recall Hazards

- Structural hazards
  - Two instructions need the same hardware
  - Half/half for REG and two copies for MEM
- Data Hazards
  - Data not ready
  - Forward/bypass (stall for loads)
- Control flow Hazards
  - Which instruction to fetch? Not known.
  - Move up br. decision, Predict not taken, delayed branch

Recap: Datapath for branch hazards



Common confusion in CPI calculation

- Ideal CPI = 1, 1/4th loads, 1/6th branches, br prediction accuracy 90% and branches resolved at the END of MEM and assume 50% loads are followed immediately by a dependent instr and by independent instr for rest
- CPI = 1 + 1/4 *1/2 *1 + 1/6 * 0.1 * 3
  - The first "1" captures no stalls for ALL instructions, so later terms are only for stalls

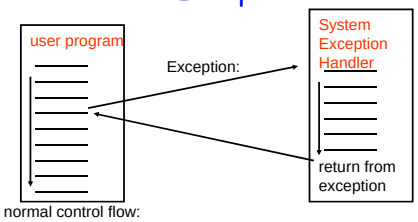
Common confusion in CPI calculation

- CPI = 1 + 1/4 *1/2 *1 + 1/6 * 0.1 * 3
- DO NOT do
  - 1 – ($\frac{1}{4}$ + 1/6) + $\frac{1}{4}$*1/2*1 + $\frac{1}{4}$*1/2 * 2 + 1/6 *0.9*1 + 1/6*0.1*4
    - correct but laborious
  - 1 + $\frac{1}{4}$*1+1/6*3
    - wrong because double-counts non-stall cases

What next?

- Exceptions
  - Multiple instructions in flight
  - PC has changed
- Advanced topics – VERY briefly
  - Superscalar, dynamically scheduled processors, etc
- Real machines
  - Intel i7 pipeline, Niagara Pipeline

Exceptions



normal control flow:
sequential, jumps, branches, calls, returns

- Exception = unprogrammed 'surprise' function call
  - OS takes action to handle the exception
    - must record the address of the offending instruction
  - returns control to user
  - must save & restore user state
- Lets OS get in/out without breaking user program – as if user program had a "virtual CPU" to itself

Interrupt, Exception, Trap?

- Interrupts
  - caused by external events (eg I/O)
  - asynchronous to program execution
  - may be handled between instructions
  - suspend user program, handle interrupt, resume user program
- Traps
  - caused by internal events
    - exceptional conditions (overflow)
    - errors (parity)
    - faults (ch. 5) – KEY mechanism in all modern computers
  - synchronous to program execution
  - condition must be remedied by the handler
  - instruction may be retried or simulated and program continued, or program may be aborted
- MIPS convention:
  - External : Interrupts
  - Internal : Exception

Exception Semantics

- MIPS architecture defines the instruction as having no effect if the instruction causes an exception.
- When we get to virtual memory (ch 5b) we will see that certain classes of exceptions must prevent the instruction from changing the machine state.
- This aspect of handling exceptions becomes complex and potentially limits performance => why it is hard
  - Precise interrupts vs Imprecise interrupts

Exceptions

- Pipeline Semantics
  - NO instruction AFTER the excepting instruction may execute
  - EVERY instruction BEFORE the excepting instruction must complete execution
    - Sounds similar to what?
  - Exception handler software saves and restores registers and other state
    - Different from 362 microcontroller

MIPS Exceptions

- All exceptions jump to same handler code
  - "Cause" register
- We consider
  - Illegal instructions
  - Arithmetic overflows
- Hardware behavior
  - Save PC of offending instruction (How? PC+4 has already been written to PC)
  - Use special register EPC(why not use $31 like jal?)
  - Set cause register appropriately (0=ILL; 1=OVF)
  - Jump to handler at fixed address

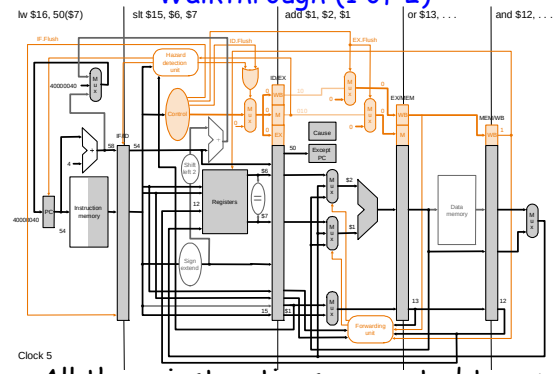
Datapath modifications

- Pipeline complications
- Which stage is exception detected?
  - Overflow?
    - In EX, squash (convert to nop) EX & earlier stages
  - Illegal Instruction?
    - In ID, squash (convert to nop) ID & earlier stages
    - Similar to RAW hazard
  - What about external interrupts?
- Overflow in instruction i, illegal instruction in instruction i+1
  - Simultaneous exceptions
  - Hardware sorting

Walk-through: Code snippet

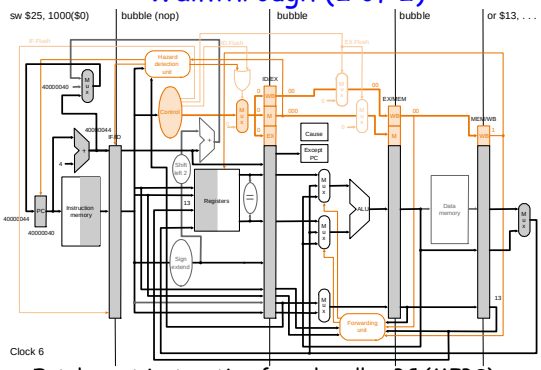
- Main Code
  - 40 sub $11, $2, $4
  - 44 and $12, $2, $5
  - 48 or $13, $2, $1
  - 4C add $1, $2, $1
  - 50 slt $15, $6, $7
- Exception Code
  - [Exception handler PC]
    - sw $25, 1000($0)

Walkthrough (1 of 2)



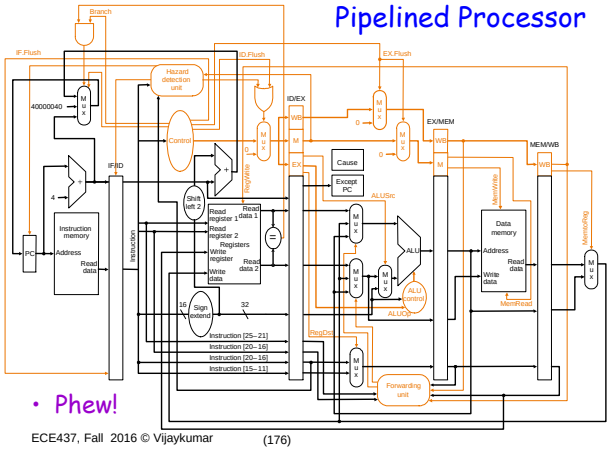
- All three instructions converted to nop

Walkthrough (2 of 2)



- Fetch next instruction from handler PC (MIPS)

Pipelined Processor



- Phew!

Midterm 1

- 10/14 Wednesday 8-10 pm SMTH 108
- Ch. 1, 2, and 4 + slides (1a,1b,2,4a,4b)
- Sample exam on BlackBoard – folder "Midterms"

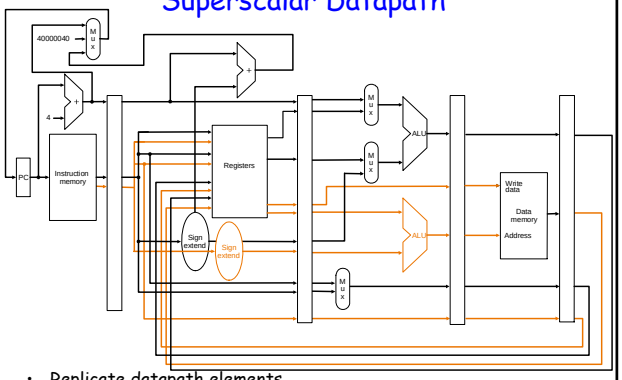
Understanding Performance

- 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
- Can we do better?
  - CPI < 1?
  - How about CPI of 0.5?

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
- Static Multiple issue datapath

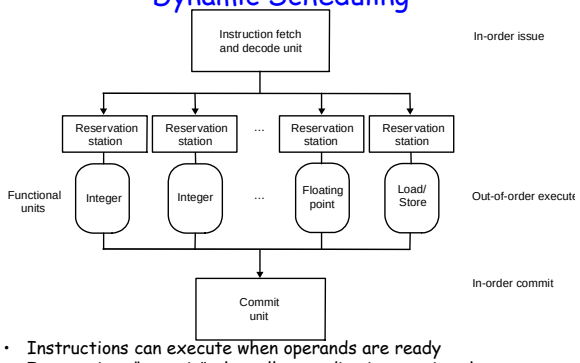
Dynamic Scheduling

- No need to suffer hazards if other useful work can be achieved
- Load hazard results in pipeline stall
  - But later instructions are ready
  - "Oh! But we cannot execute instructions out of order" – Not true! Modern CPUs do it all the time!!

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

  - Execute sub,slti, & later instrs while lw has not completed!

Dynamic Scheduling



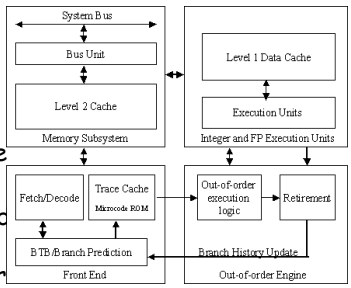
- Instructions can execute when operands are ready
- Instructions "commit" when all preceding instructions have committed

Real machines

- Let's examine Pentium 4, Core 2, i7
  - Microarchitecture more or less stable
  - Technology has improved
  - Also ARM A3, Sun Niagara

Pentium 4 on 0.18 micron

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



Pentium 4 pipeline

Basic Pentium® III Processor Misprediction Pipeline

1	2	3	4	5	6	7	8	9	10
Fetch	Fetch	Decode	Decode	Decode	Rename	ROB Rd	Rdy/Sch	Dispatch	Exec

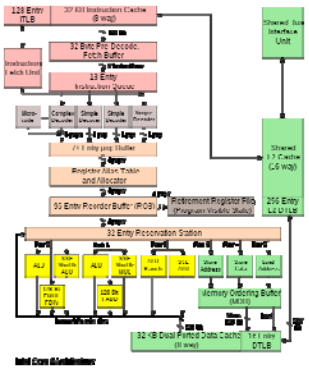
Basic Pentium® 4 Processor Misprediction Pipeline

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
TC Nxt IP		TC Fetch		Drive	Alloc	Rename		Que	Sch	Sch	Sch	Disp	Disp	RF	RF	Ex	Flgs	Br Ck	Drive

- One specific pipeline (misprediction)

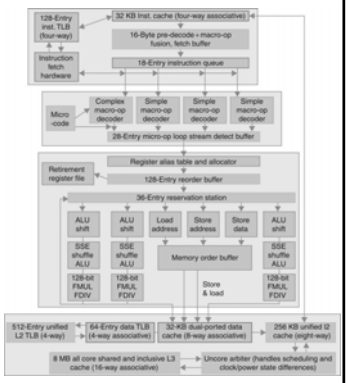
Core 2

- 45nm
- Multiple decodes
- 14 stage pipeline
  - (went as high as ~31 in Pentium 4 line)
  - Many other considerations
    - Pipelining for yield
- Source: Wikipedia

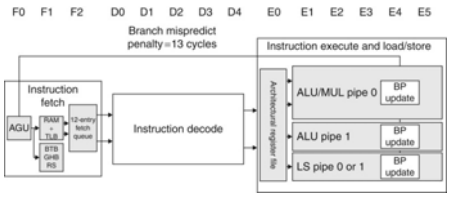


i7

- 6 uops/cycle
- 14 stage pipeline
  - 17 cycles br penalty
  - 48 loads and 32 stores in-flight



ARM A3



- 3-cycle fetch
  - Addr Generation Unit uses BTB
- 5-cycle decode and 6-cycle EX

Sun Niagara

- Not too dissimilar from 437
- 4 threads
- Eight such processors on a chip
- March/April 2005 Issue of IEEE MICRO

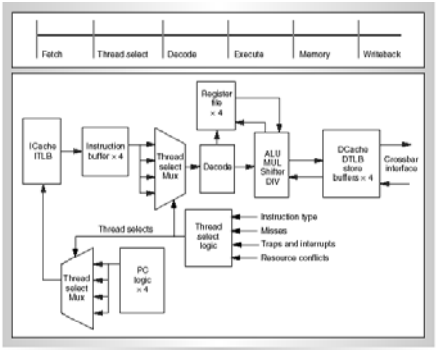


Figure 3. Sparc pipeline block diagram. Four threads share a six-stage single-issue pipeline with local instruction and data caches. Communication with the rest of the machine occurs through the crossbar interface.

Summary

- Exceptions
  - Know how to handle the easy cases
    - What to squash, what not to
  - Know how complicated exceptions can be
- Read Chapter 4 NOW
  - Maximize impact
  - Study while lecture material is "warm"
    - 2-3 hours now vs. 8-12 hours later