CS 61C: Great Ideas in Computer Architecture

Lecture 13: *Pipelining*

Krste Asanović & Randy Katz

http://inst.eecs.berkeley.edu/~cs61c/fa17

Agenda

- **RISC-V Pipeline**
- Pipeline Control
- Hazards
  - Structural
  - Data
    - R-type instructions
    - Load
  - Control
- Superscalar processors

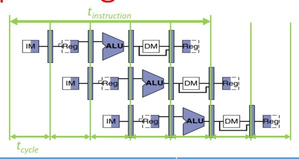
Recap: Pipelining with RISC-V

instruction sequence

add t0, t1, t2

or t3, t4, t5

sll t6, t0, t3



	Single Cycle	Pipelining
Timing	t_{step} = 100 … 200 ps	t_{cycle} = 200 ps
	Register access only 100 ps	All cycles same length
Instruction time, $t_{instruction}$	= t_{cycle} = 800 ps	1000 ps
Clock rate, f_s	1/800 ps = 1.25 GHz	1/200 ps = 5 GHz
Relative speed	1 x	4 x

RISC-V Pipeline

instruction sequence

add t0, t1, t2

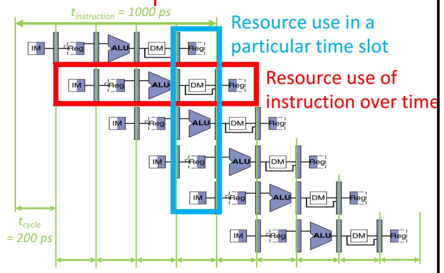
or t3, t4, t5

slt t6, t0, t3

sw t0, 4(t3)

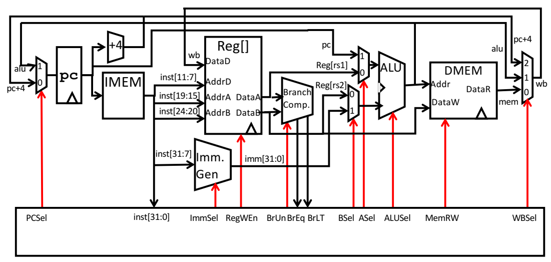
lw t0, 8(t3)

addi t2, t2, 1

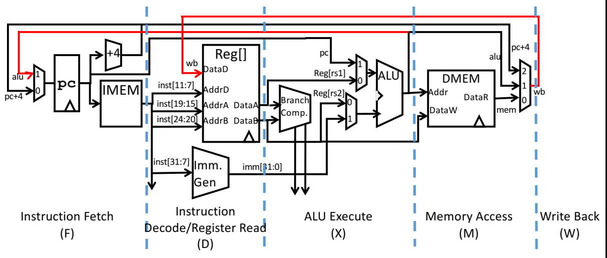


$t_{instruction}$ = 1000 ps
Resource use in a particular time slot
Resource use of instruction over time
t_{cycle} = 200 ps
IM
Reg
ALU
DM
Reg

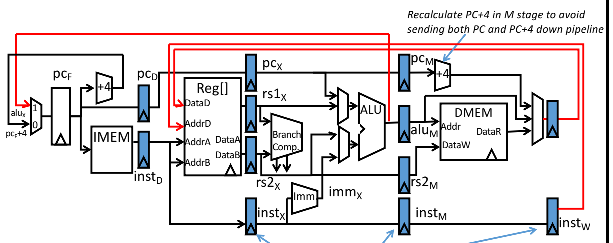
Single-Cycle RISC-V RV32I Datapath



Pipelining RISC-V RV32I Datapath

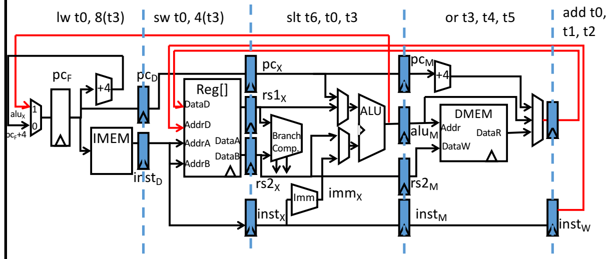


Pipelined RISC-V RV32I Datapath



Must pipeline instruction along with data, so control operates correctly in each stage

Each stage operates on different instruction



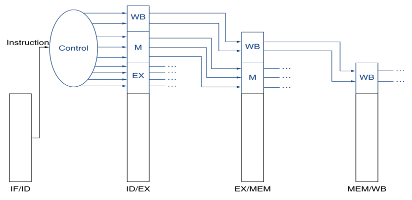
Pipeline registers separate stages, hold data for each instruction in flight

Agenda

- RISC-V Pipeline
- **Pipeline Control**
- Hazards
  - Structural
  - Data
    - R-type instructions
    - Load
  - Control
- Superscalar processors

Pipelined Control

- Control signals derived from instruction
  - As in single-cycle implementation
  - Information is stored in pipeline registers for use by later stages



Hazards Ahead

Agenda

- RISC-V Pipeline
- Pipeline Control
- Hazards
  - **Structural**
  - Data
    - R-type instructions
    - Load
  - Control
- Superscalar processors

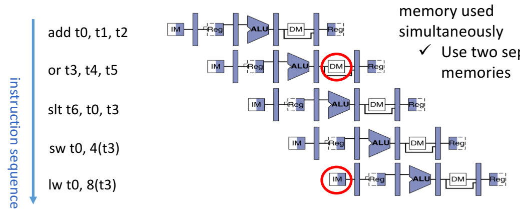
Structural Hazard

- **Problem:** Two or more instructions in the pipeline compete for access to a single physical resource
- **Solution 1:** Instructions take it in turns to use resource, some instructions have to stall
- **Solution 2:** Add more hardware to machine
- Can always solve a structural hazard by adding more hardware

Regfile Structural Hazards

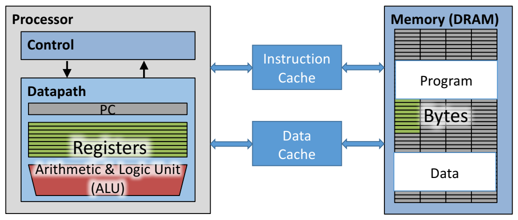
- Each instruction:
  - can read up to two operands in decode stage
  - can write one value in writeback stage
- Avoid structural hazard by having separate "ports"
  - two independent read ports and one independent write port
- Three accesses per cycle can happen simultaneously

Structural Hazard: Memory Access



- Instruction and data memory used simultaneously
  - ✓ Use two separate memories

Instruction and Data Caches



Caches: small and fast "buffer" memories

Structural Hazards – Summary

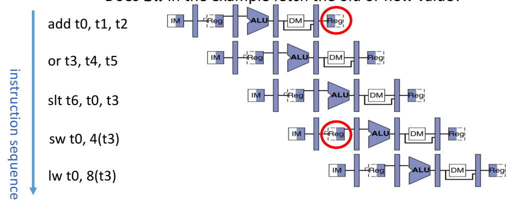
- Conflict for use of a resource
- In RISC-V pipeline with a single memory
  - Load/store requires data access
  - Without separate memories, instruction fetch would have to *stall* for that cycle
    - All other operations in pipeline would have to wait
- Pipelined datapaths require separate instruction/data memories
  - Or separate instruction/data caches
- RISC ISAs (including RISC-V) designed to avoid structural hazards
  - e.g. at most one memory access/instruction

Agenda

- RISC-V Pipeline
- Pipeline Control
- Hazards
  - Structural
  - **Data**
    - **R-type instructions**
    - Load
  - Control
- Superscalar processors

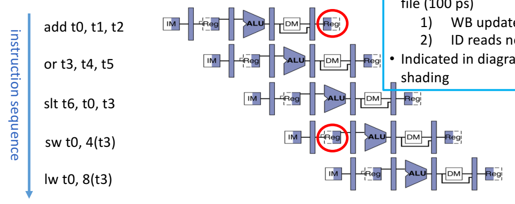
Data Hazard: Register Access

- Separate ports, but what if write to same value as read?
- Does **sw** in the example fetch the old or new value?



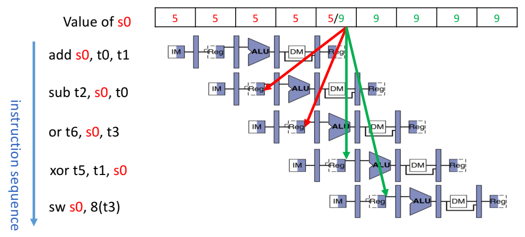
Register Access Policy

- Exploit high speed of register file (100 ps)
  1) WB updates value
  2) ID reads new value
- Indicated in diagram by shading



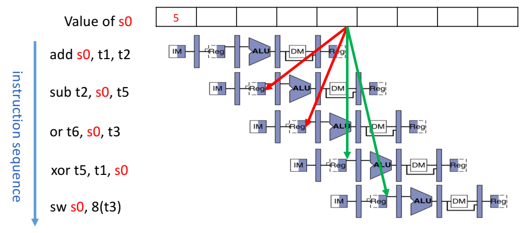
Might not always be possible to write then read in same cycle, especially in high-frequency designs. Check assumptions in any question.

Data Hazard: ALU Result



Without some fix, **sub** and **or** will calculate wrong result!

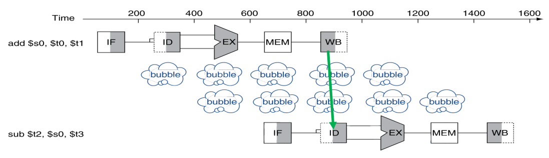
Data Hazard: ALU Result



Without some fix, **sub** and **or** will calculate wrong result!

Solution 1: Stalling

- Problem: Instruction depends on result from previous instruction
  - add s0, t0, t1
    sub t2, s0, t3

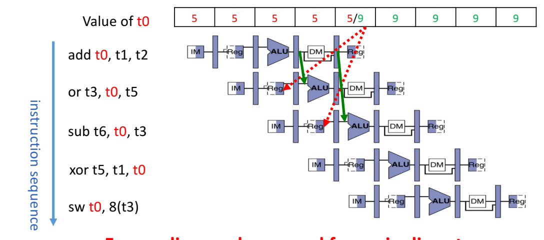


- Bubble:
  - effectively NOP: affected pipeline stages do "nothing"

Stalls and Performance

- Stalls reduce performance
  - But stalls are required to get correct results
- Compiler can arrange code to avoid hazards and stalls
  - Requires knowledge of the pipeline structure

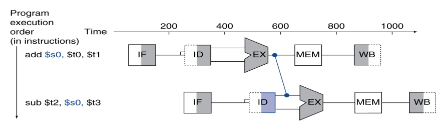
Solution 2: Forwarding



Forwarding: grab operand from pipeline stage, rather than register file

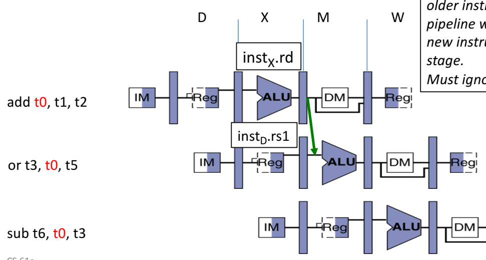
Forwarding (aka Bypassing)

- Use result when it is computed
  - Don't wait for it to be stored in a register
  - Requires extra connections in the datapath

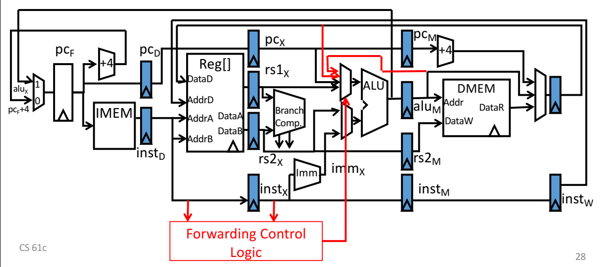


1) Detect Need for Forwarding (example)

Compare destination of older instructions in pipeline with sources of new instruction in decode stage.
Must ignore writes to x0!



Forwarding Path



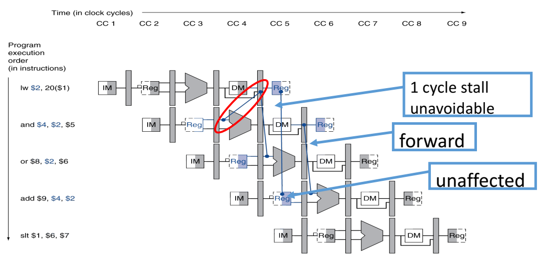
Administrivia

- Project 1 Part 2 due next Monday
  - Project Party this Wednesday 7-9pm in Cory 293
- HW3 will be released by Friday
- Midterm 1 regrades due tonight
- Guerrilla Session tonight 7-9pm in Cory 293

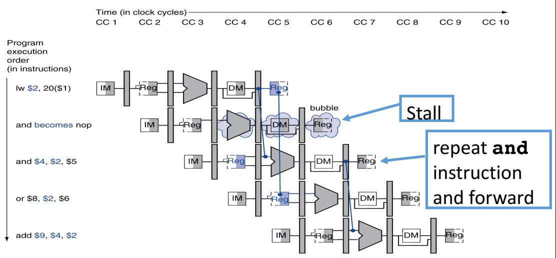
Agenda

- RISC-V Pipeline
- Pipeline Control
- Hazards
  - Structural
  - Data
    - R-type instructions
    - **Load**
  - Control
- Superscalar processors

Load Data Hazard



Stall Pipeline



`lw` Data Hazard

- Slot after a load is called a *load delay slot*
  - If that instruction uses the result of the load, then the hardware will stall for one cycle
  - Equivalent to inserting an explicit **`nop`** in the slot
    - except the latter uses more code space
  - Performance loss
- Idea:
  - Put unrelated instruction into load delay slot
  - No performance loss!

Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction!
- RISC-V code for `D=A+B; E=A+C;`

```
        Original Order:
        lw   t1, 0(t0)
        lw   t2, 4(t0)
Stall!  add  t3, t1, t2
        sw   t3, 12(t0)
        lw   t4, 8(t0)
Stall!  add  t5, t1, t4
        sw   t5, 16(t0)
```

13 cycles

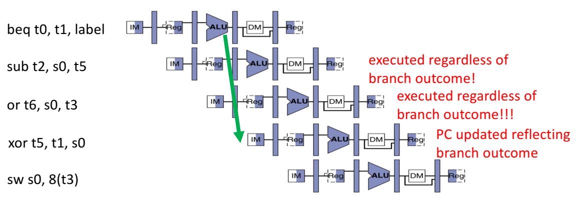
```
Alternative:
lw   t1, 0(t0)
lw   t2, 4(t0)
lw   t4, 8(t0)
add  t3, t1, t2
sw   t3, 12(t0)
add  t5, t1, t4
sw   t5, 16(t0)
```

11 cycles

Agenda

- RISC-V Pipeline
- Pipeline Control
- Hazards
  - Structural
  - Data
    - R-type instructions
    - Load
  - **Control**
- Superscalar processors

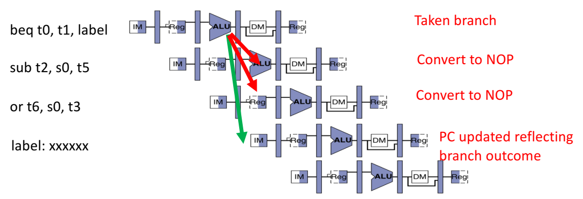
Control Hazards


beq t0, t1, label
sub t2, s0, t5
or t6, s0, t3
xor t5, t1, s0
sw s0, 8(t3)
IM
Reg
ALU
DM
executed regardless of branch outcome!
executed regardless of branch outcome!!!
PC updated reflecting branch outcome

Observation

- If branch not taken, then instructions fetched sequentially after branch are correct
- If branch or jump taken, then need to flush incorrect instructions from pipeline by converting to NOPs

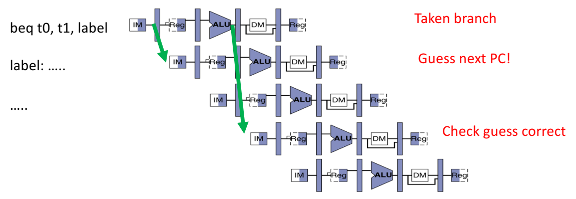
Kill Instructions after Branch if Taken



Reducing Branch Penalties

- Every taken branch in simple pipeline costs 2 dead cycles
- To improve performance, use “branch prediction” to guess which way branch will go earlier in pipeline
- Only flush pipeline if branch prediction was incorrect

Branch Prediction



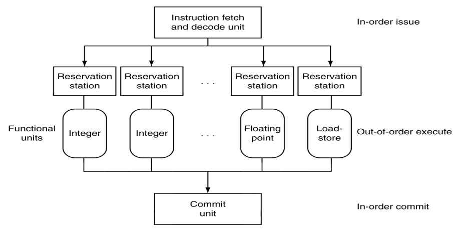
Agenda

- RISC-V Pipeline
- Pipeline Control
- Hazards
  - Structural
  - Data
    - R-type instructions
    - Load
  - Control
- **Superscalar processors**

Increasing Processor Performance

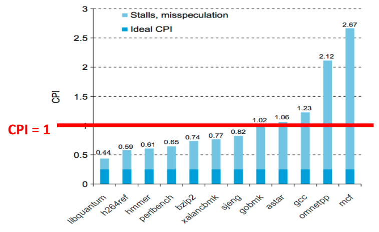
1. Clock rate
   - Limited by technology and power dissipation
2. Pipelining
   - “Overlap” instruction execution
   - Deeper pipeline: 5 => 10 => 15 stages
     - Less work per stage → shorter clock cycle
     - But more potential for hazards (CPI > 1)
3. Multi-issue ”super-scalar” processor
   - Multiple execution units (ALUs)
     - Several instructions executed simultaneously
     - CPI < 1 (ideally)

Superscalar Processor



P&H p. 340

Benchmark: CPI of Intel Core i7



CPI of Intel Core i7 920 running SPEC2006 integer benchmarks.

P&H p. 350

In Conclusion

- Pipelining increases throughput by overlapping execution of multiple instructions
- All pipeline stages have same duration
  - Choose partition that accommodates this constraint
- Hazards potentially limit performance
  - Maximizing performance requires programmer/compiler assistance
  - E.g. Load and Branch delay slots
- Superscalar processors use multiple execution units for additional instruction level parallelism
  - Performance benefit highly code dependent

Extra Slides

Pipelining and ISA Design

- RISC-V ISA designed for pipelining
  - All instructions are 32-bits
    - Easy to fetch and decode in one cycle
    - Versus x86: 1- to 15-byte instructions
  - Few and regular instruction formats
    - Decode and read registers in one step
  - Load/store addressing
    - Calculate address in 3rd stage, access memory in 4th stage
  - Alignment of memory operands
    - Memory access takes only one cycle

Superscalar Processor

- Multiple issue "superscalar"
  - Replicate pipeline stages ⇒ multiple pipelines
  - Start multiple instructions per clock cycle
  - CPI < 1, so use Instructions Per Cycle (IPC)
  - E.g., 4GHz 4-way multiple-issue
    - 16 BIPS, peak CPI = 0.25, peak IPC = 4
  - Dependencies reduce this in practice
- "Out-of-Order" execution
  - Reorder instructions dynamically in hardware to reduce impact of hazards
- *CS152 discusses these techniques!*