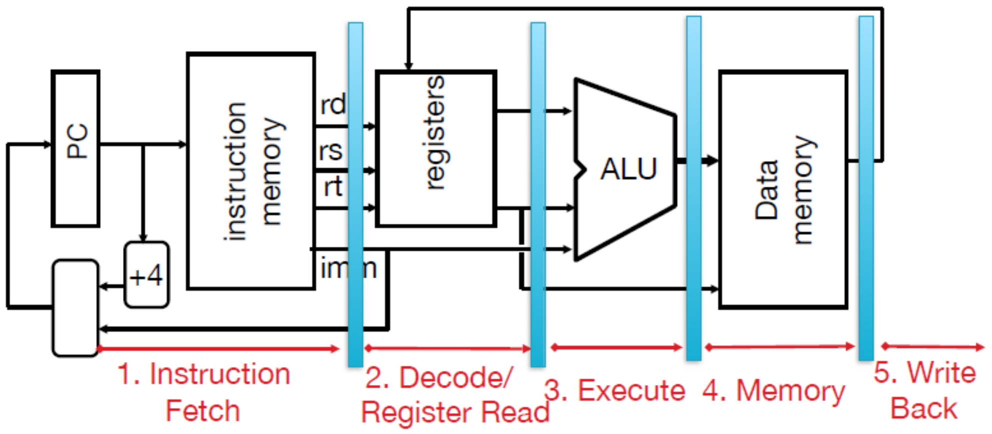
CSM CS61C Note #6: Pipelining, Pipelining Hazards

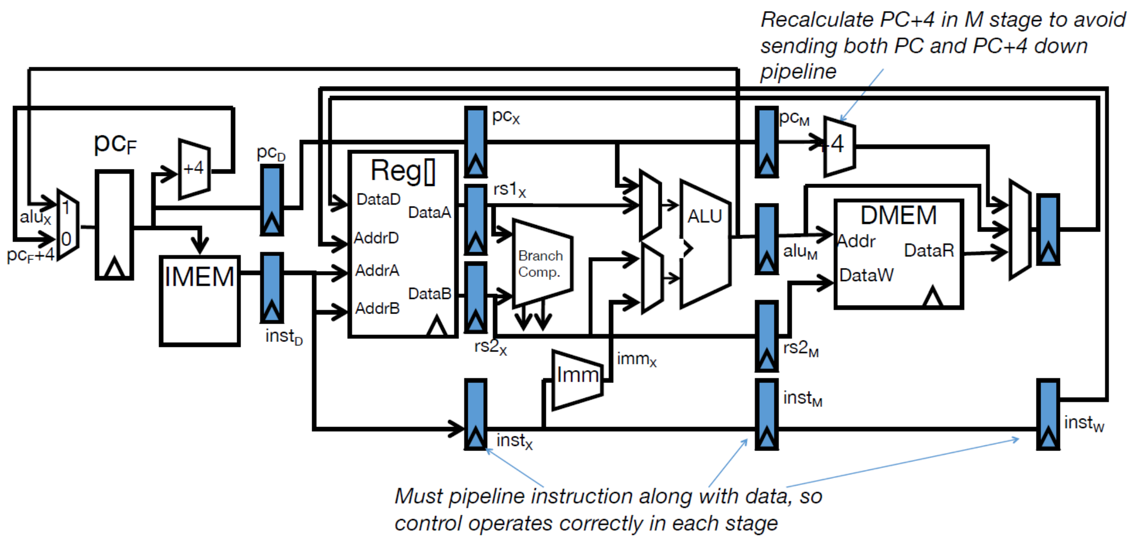
Anthony Han

1 Pipelined RISC-V Datapath

In a single-cycle datapath, only one component is doing the computations. To increase the clock frequency, we pipeline the circuit so that multiple components are in use at the same time, using **registers between stages to hold information produced in previous cycles:**



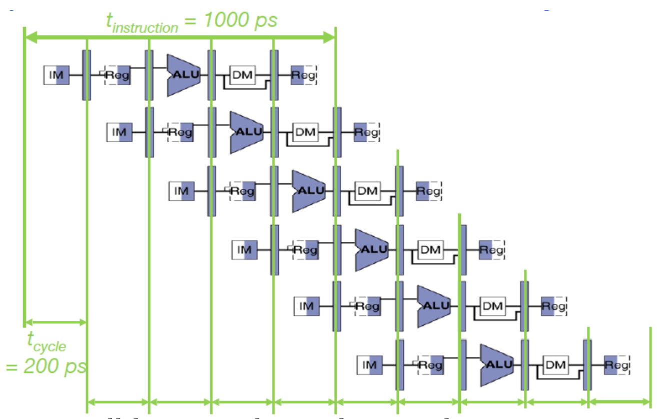
And we have our pipelined 5-stage RISC-V processor:


Recalculate PC+4 in M stage to avoid sending both PC and PC+4 down pipeline
pcF
+4
pcD
aluX
pcF+4
1
0
IMEM
instD
Reg[]
DataD
DataA
AddrD
AddrA
DataB
AddrB
pcX
rs1X
Branch Comp.
rs2X
Imm
immX
instX
ALU
pcM
+4
aluM
rs2M
instM
DMEM
Addr
DataR
DataW
instW
Must pipeline instruction along with data, so control operates correctly in each stage

- **Increases throughput** - multiple tasks operating simultaneously using different resources.
- **Never improves latency; sometimes worse**.
- **Potential speedup equals the number of pipeline stages.**
- **Time to "fill" and "drain" pipeline reduces speedup**; approaches potential speedup as number of instructions increases.

Phase	Pictogram	t_{step} Serial	t_{cycle} Pipelined
Instruction Fetch	IM	200 ps	200 ps
Reg Read	Reg	100 ps	200 ps
ALU	ALU	200 ps	200 ps
Memory	DM	200 ps	200 ps
Register Write	Reg	100 ps	200 ps
$t_{instruction}$	IM Reg ALU DM Reg	**800 ps**	**1000 ps**

```
add  t0, t1, t2

or   t3, t4, t5

sll  t6, t0, t3

sw   t0, 4(t3)

lw   t0, 8(t3)

addi t2, t2, 1
```



Starting from the fifth cycle, all five stages will be at work simultaneously.

2 Pipelining Hazards

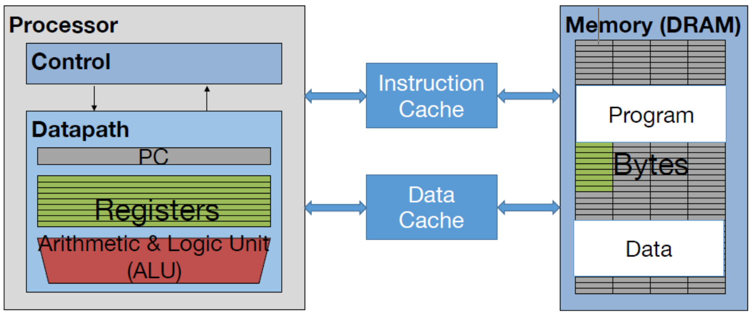
A **hazard** is a situation that **prevents us from starting the next instruction in the next clock cycle.**

Structural Hazard

- Two or more **instructions** in the pipeline **race for access to a single physical resource.**
- Two possible solutions:
  - Solution 1: Instructions take turns to use resource; **stall some instructions.**
  - Solution 2: **Add more hardware.** We can always solve a structural hazard by adding more hardware.
- **Regfile Structural Hazards**
  - Each instruction can read up to two operands in decode stage and write one value in writeback stage.

  - We can **avoid structural hazard by having separate "ports"**: two independent read ports and one independent write port.
  - **Two reads and one write can happen simultaneously in one cycle.**

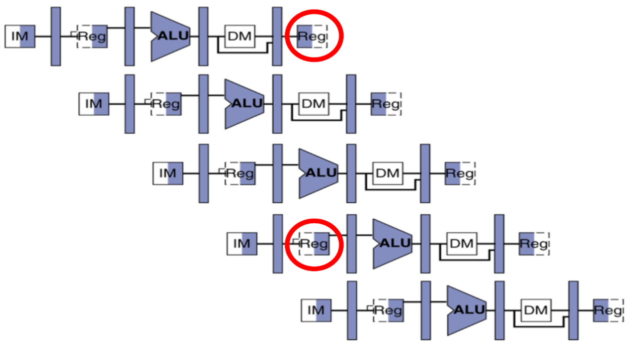
- **Memory Structural Hazards**
  - Instruction and data memory used simultaneously.
  - We could **just use two separate memories.**
  - Instruction and data caches:



- RISC ISAs are **designed to avoid structural hazards** - at most 1 memory access/instruction.

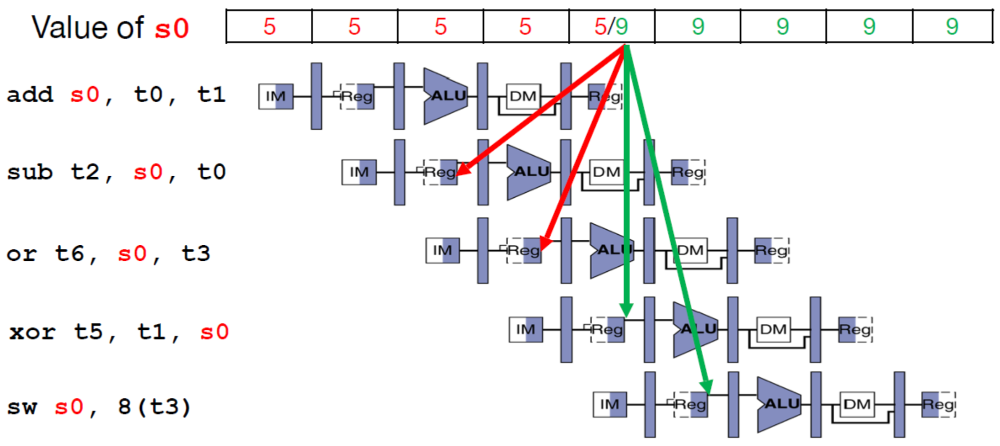
Data Hazard

- **Data dependency between instructions.**
- **Data Hazard: Register Access**
  - Although we have separate ports for read and write, but when two pipelined instructions **simultaneously read and write to a register**, the read may have ambiguous behavior.



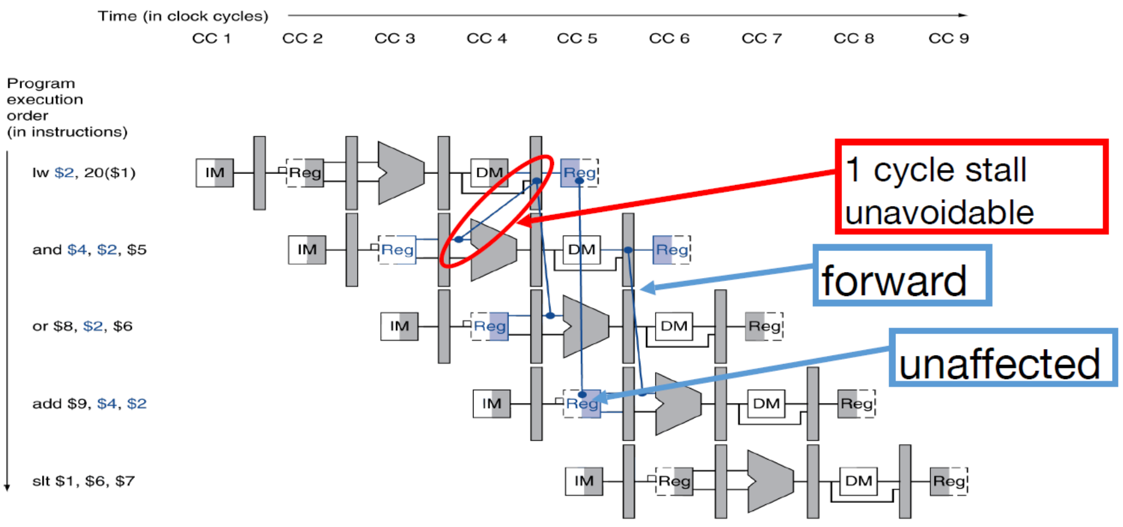
  - We could **exploit high speed of register file** ($\approx$ 100 ps) to fit everything in a clock cycle ($\approx$ 200 ps): **WB updates value first, then ID reads new value in.**
  - We note it might not always be possible to write then read in the same cycle, especially in high-frequency designs.

- **Data Hazard: ALU Result**



Without some fix, sub and or will give wrong results as these lines are dependent on the s0 value generated in the add instruction!

- **Data Hazard: Load Word**



We note that in the example above, x2 is used straight after it being loaded from memory.

- Slot after a load is called a **load delay slot. If that instruction uses the result of the load, then the hardware will have to stall for one cycle.**
- To deal with this, we could
  * **Put unrelated instruction into load delay slot** to avoid use of load result in the next instruction.
  * Consider the RISC-V code for C code:

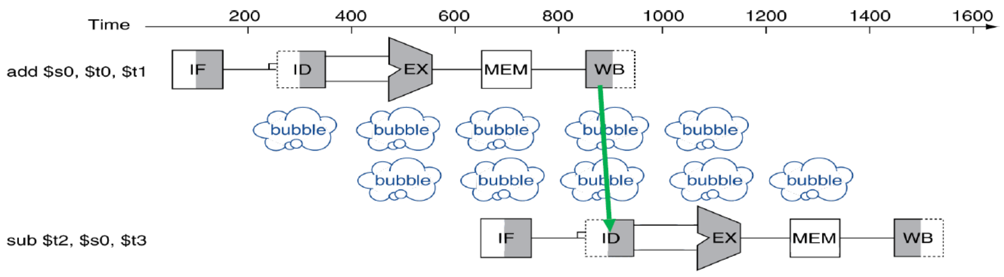
```
A[3] = A[0] + A[1];
A[4] = A[0] + A[2];
```

```
# Original Order - 9 Cycles
lw  t1, 0(t0)
lw  t2, 4(t0)
add t3, t1, t2 # Stall!
sw  t3, 12(t0)
lw  t4, 8(t0)
add t5, t1, t4 # Stall!
sw  t5, 16(t0)
```

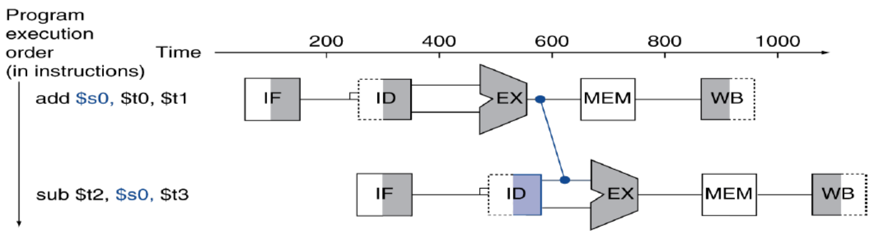
```
# Alternative - 7 Cycles
lw  t1, 0(t0)
lw  t2, 4(t0)
lw  t4, 8(t0)
add t3, t1, t2 # Move!
sw  t3, 12(t0)
add t5, t1, t4 # Move!
sw  t5, 16(t0)
```

- Two possible solutions:
  - Solution 1: **Stalling**
    - In the following code block, an instruction depends on the result from a previous instruction:

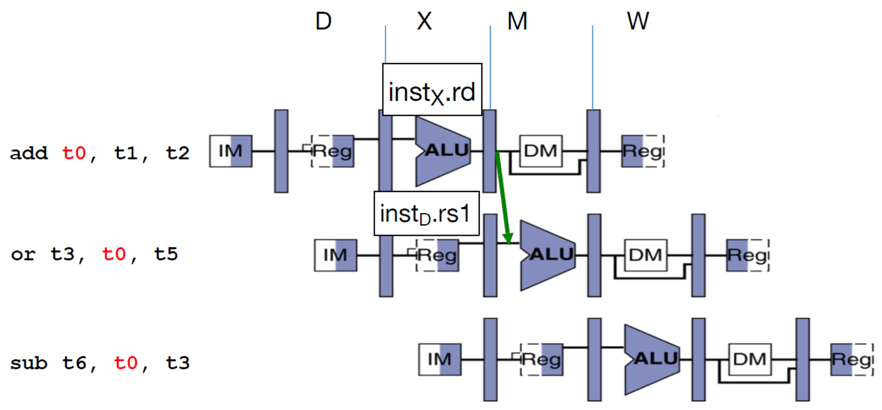
      ```
      add s0, t0, t1
      sub t2, s0, t3
      ```

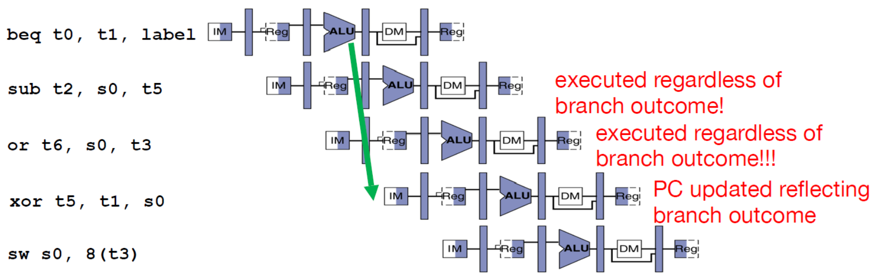
      Here, we **fill in NOPs** between the two instructions; the affected pipeline stages would simply do "nothing."
    - **Stalls reduce performance;** but are **sometimes required to get correct results.**
    - **Compiler can rearrange code or insert NOPs (writes to register x0) to avoid hazards and stalls.** This, however, requires knowledge of the pipeline structure.
  - Solution 2: **Forwarding** (or Bypassing)
    - **Grab operand from pipeline stage when it is computed.**
      - Do not wait for it to be stored in a register.
      - **Requires extra connections in the datapath.**

    - To detect need for forwarding, we **compare the destination register of older instructions in the pipeline with the source registers of new instruction in decode stage.**



Control Hazards


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

- If branch is not taken, then instructions fetched sequentially after branch should remain.
- **If branch is taken,** then we need to **flush/kill incorrect instructions from pipeline by converting them to NOPs.** Thus every taken branch in our simple pipeline costs 2 dead cycles.
- To improve performance, **use "branch prediction" to guess which way branch will go** earlier in the pipeline.
  - If branch has been seen before, predict as what you did the last time.
  - If branch has not been seen before, assume forward branches are not taken and backward branches are taken.
  - Update state on predictor with results of branch when it is finally calculated.
  - **Only flush pipeline if branch prediction was incorrect.**