Pipelining (part 2)

Chapter 6

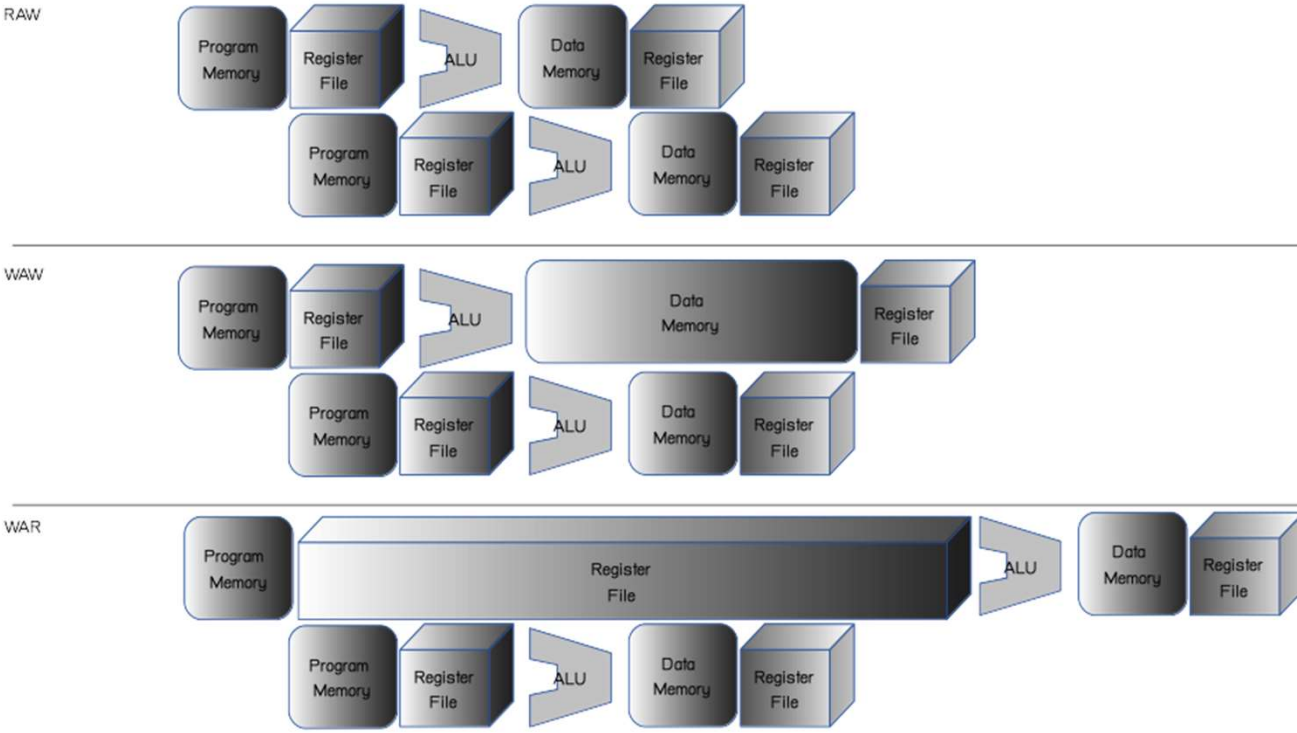
Pipelining (part 2)

- ★ Data Hazard revisit
  - RAW
  - WAW
  - WAR
- ★ Designing a Pipelined Processor
  - Hazard Detection
  - Forwarding Circuit
- ★ Compiler avoiding hazard

Data Hazard revisit

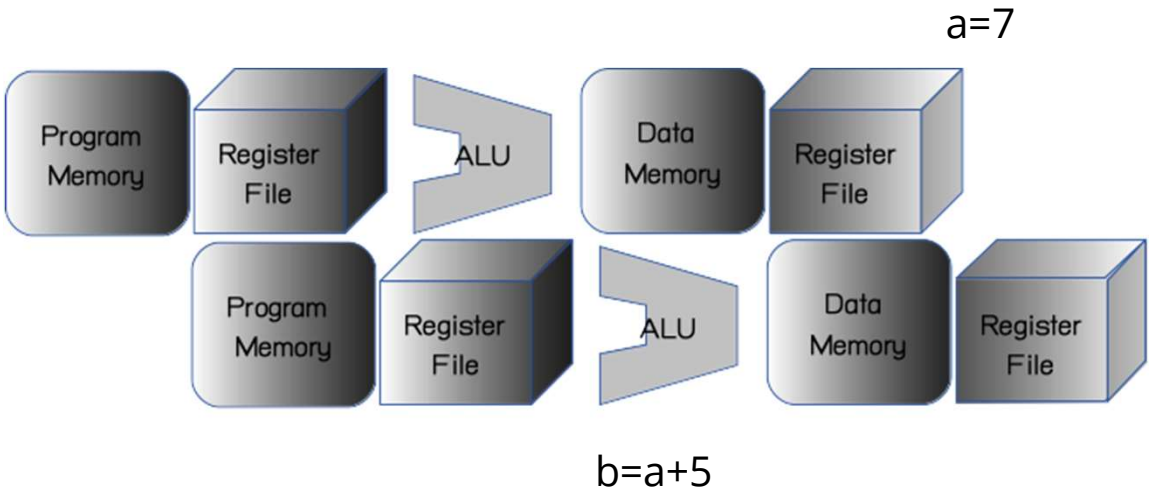
- ★ Read after Write (RAW)
- ★ Write after Write (WAW)
- ★ Write after Read (WAR)

- ★ Why there is not Read after Read?



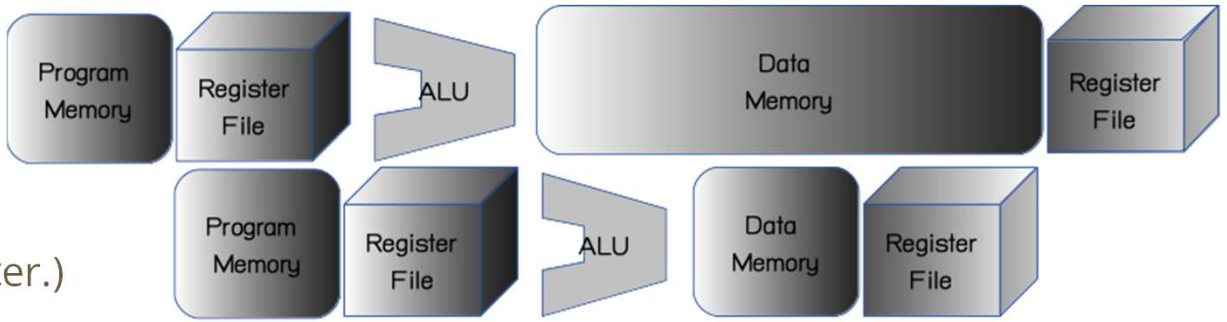
Read after Write

- ★ Consider the following code:
  - int a, b;
  - a = 7;
  - b = a + 5;
- ★ Solutions: Forwarding can solve most of the time.



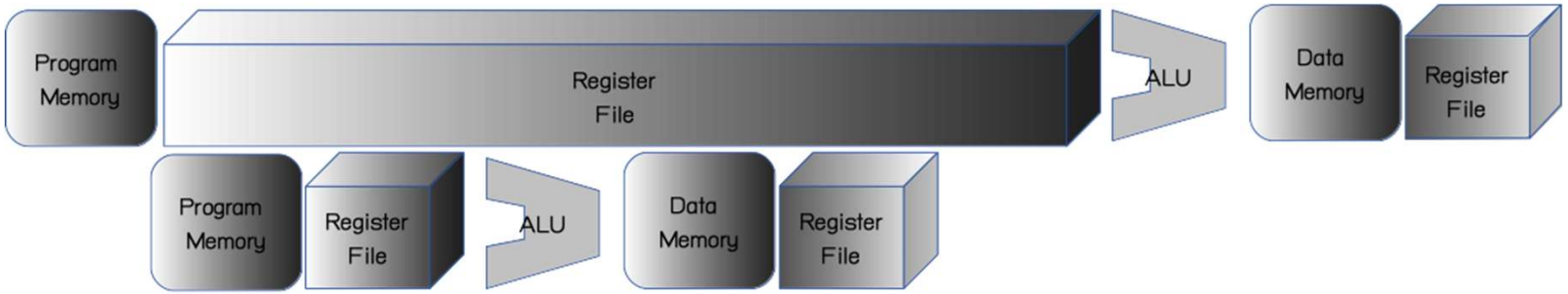
Write after Write

- Consider the following code:
  - a = data[1000];
  - a = 10;
  - b = a + 3;
- data[1000] may take several cycles (due to memory speed).
- Solutions:
  - Can avoid by design.
  - Software solution?
  - Out-of-order execution? (We will learn about this later.)



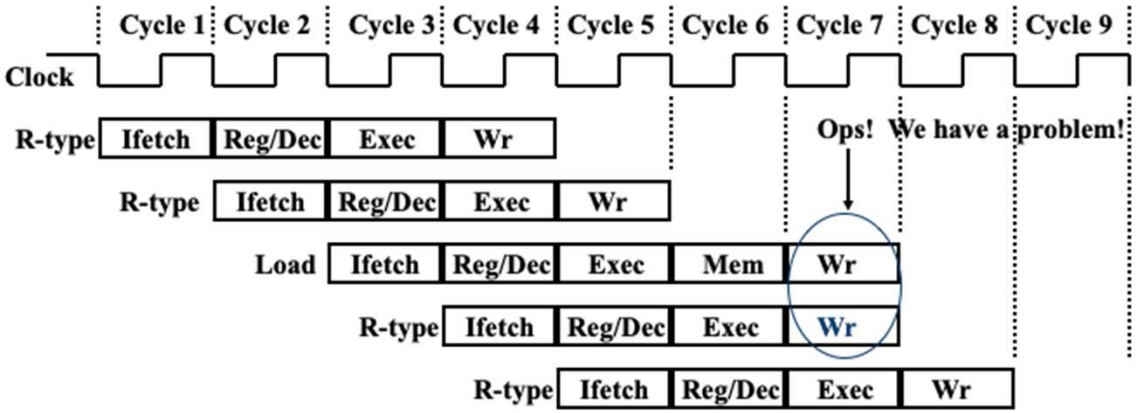
Write after Read

- ★ Hardly found in modern architecture.
- ★ Consider the following code:
  - a=sin(x);
  - x=10;
- ★ Assuming that sin function need special preparation. By the time we read x, it may be assigned.
- ★ Can avoid by design.



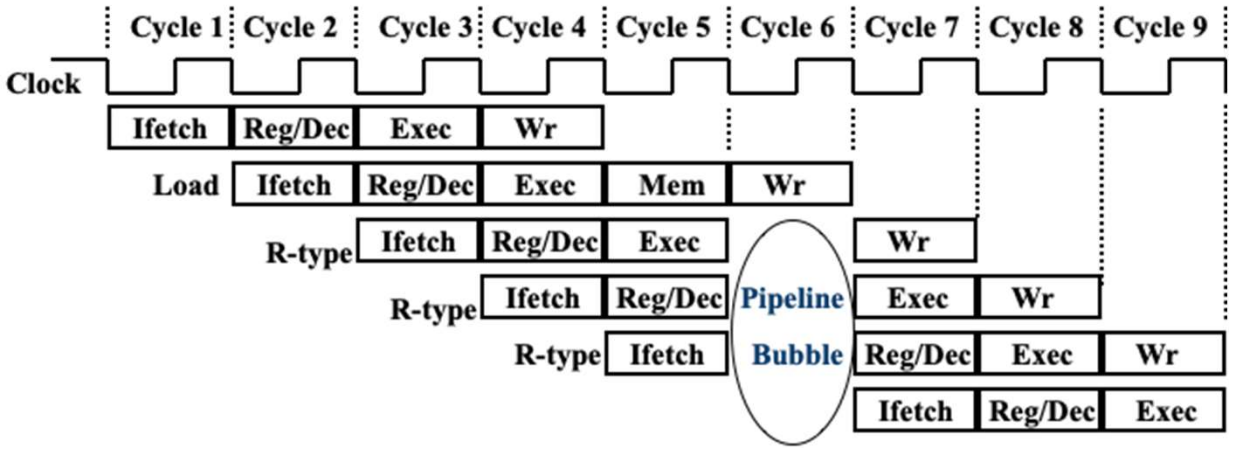
Mixing R-type with Load

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



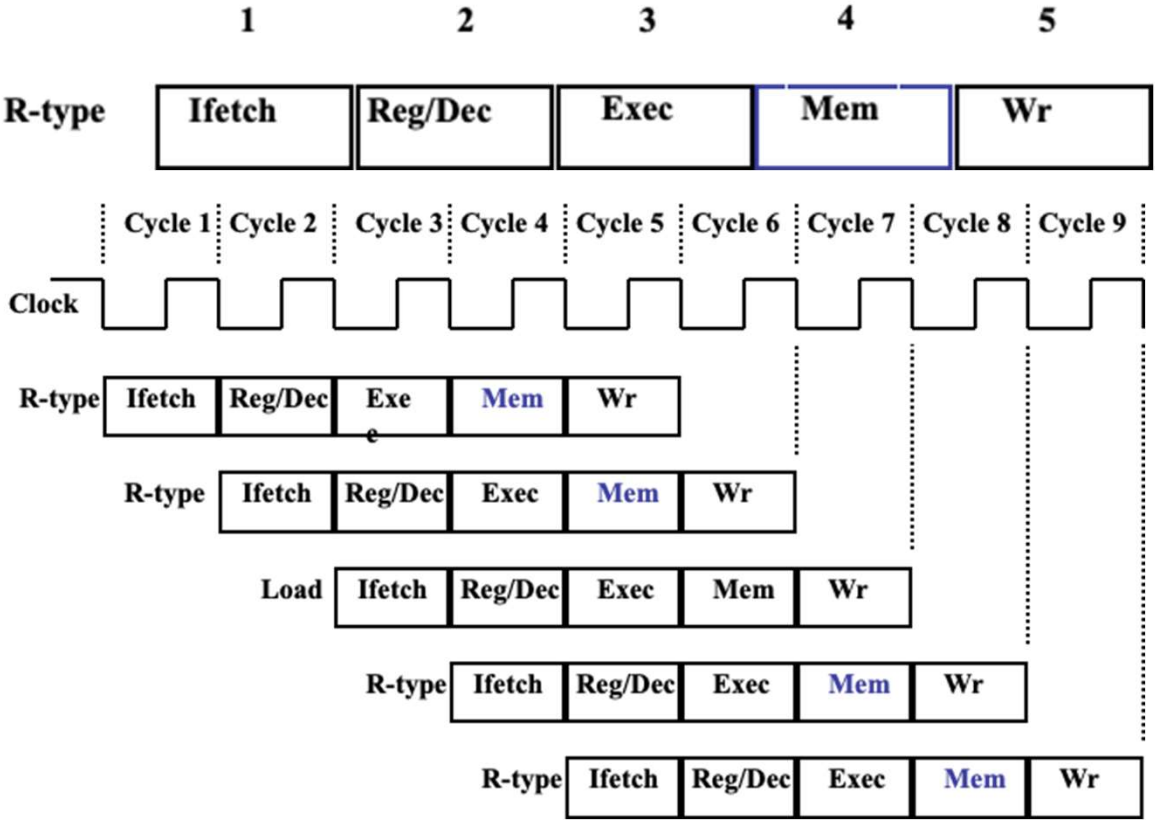
Stall Cycle by Inserting Bubble

- ★ 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!



Delay R-type write by one cycle

- ★ 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.

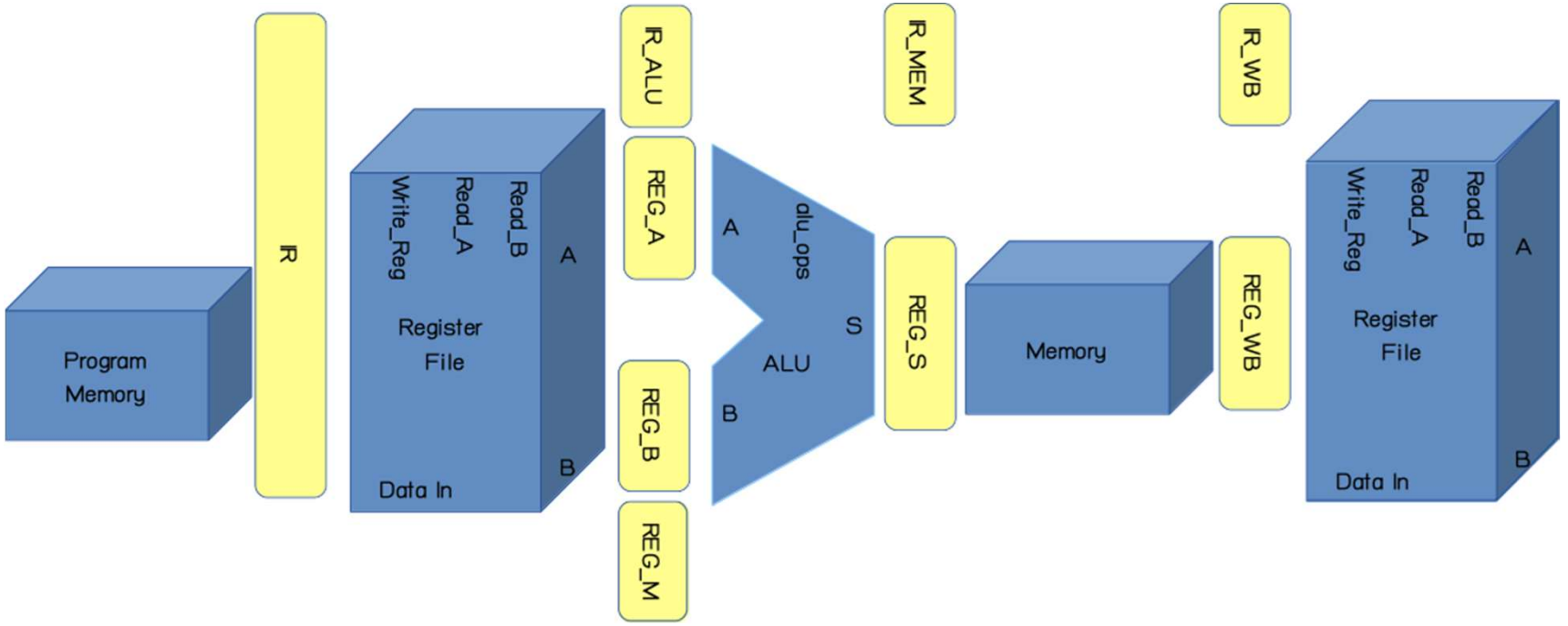


Designing a Pipelined Processor

- ★ Go back and examine your datapath and control diagram
- ★ associated resources with states
- ★ ensure that flows do not conflict, or figure out how to resolve
- ★ assert control in appropriate stage

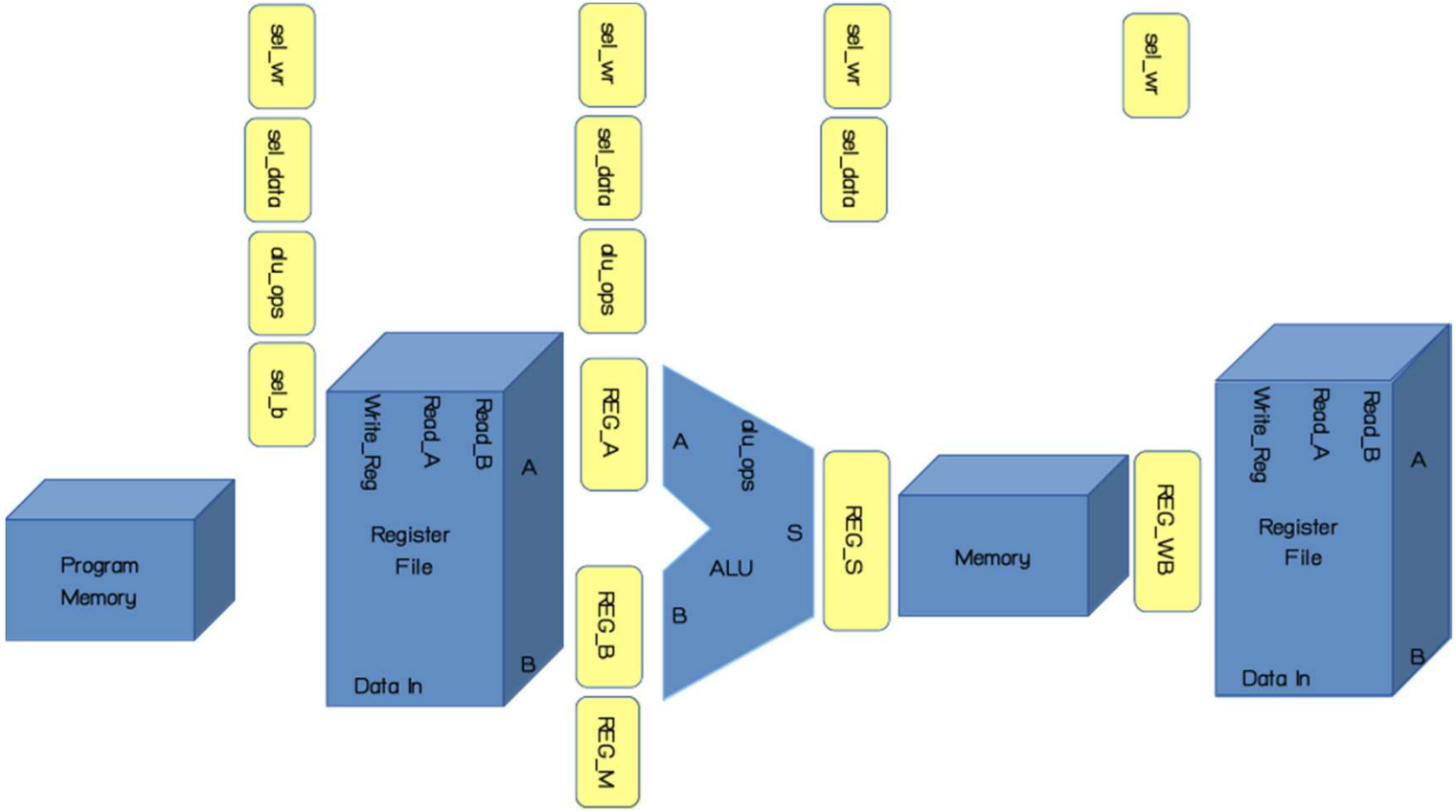
Datapath for Pipelined Processor

★ Each step has different instruction

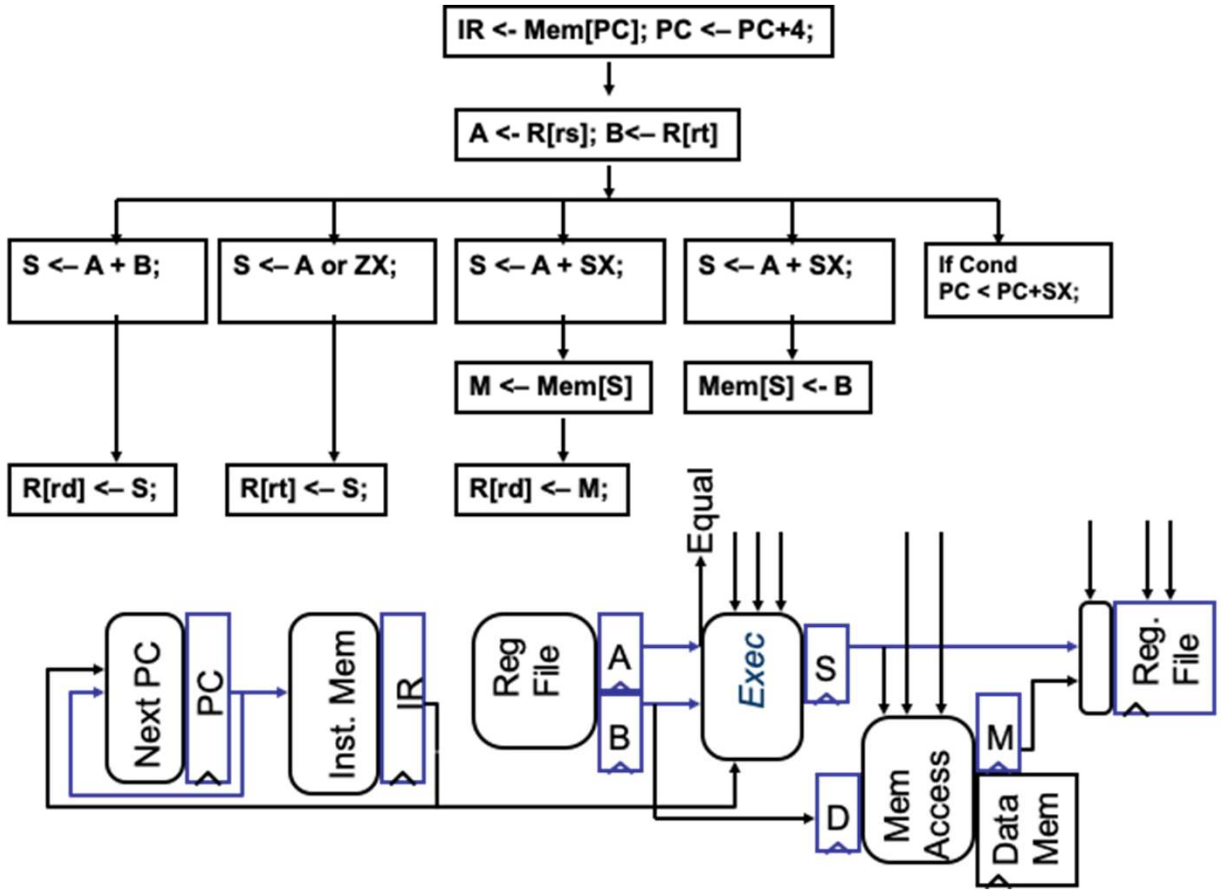


Unified Decoder

★ Alternatively, we can pass control signals only.



Datapath and Control Signal

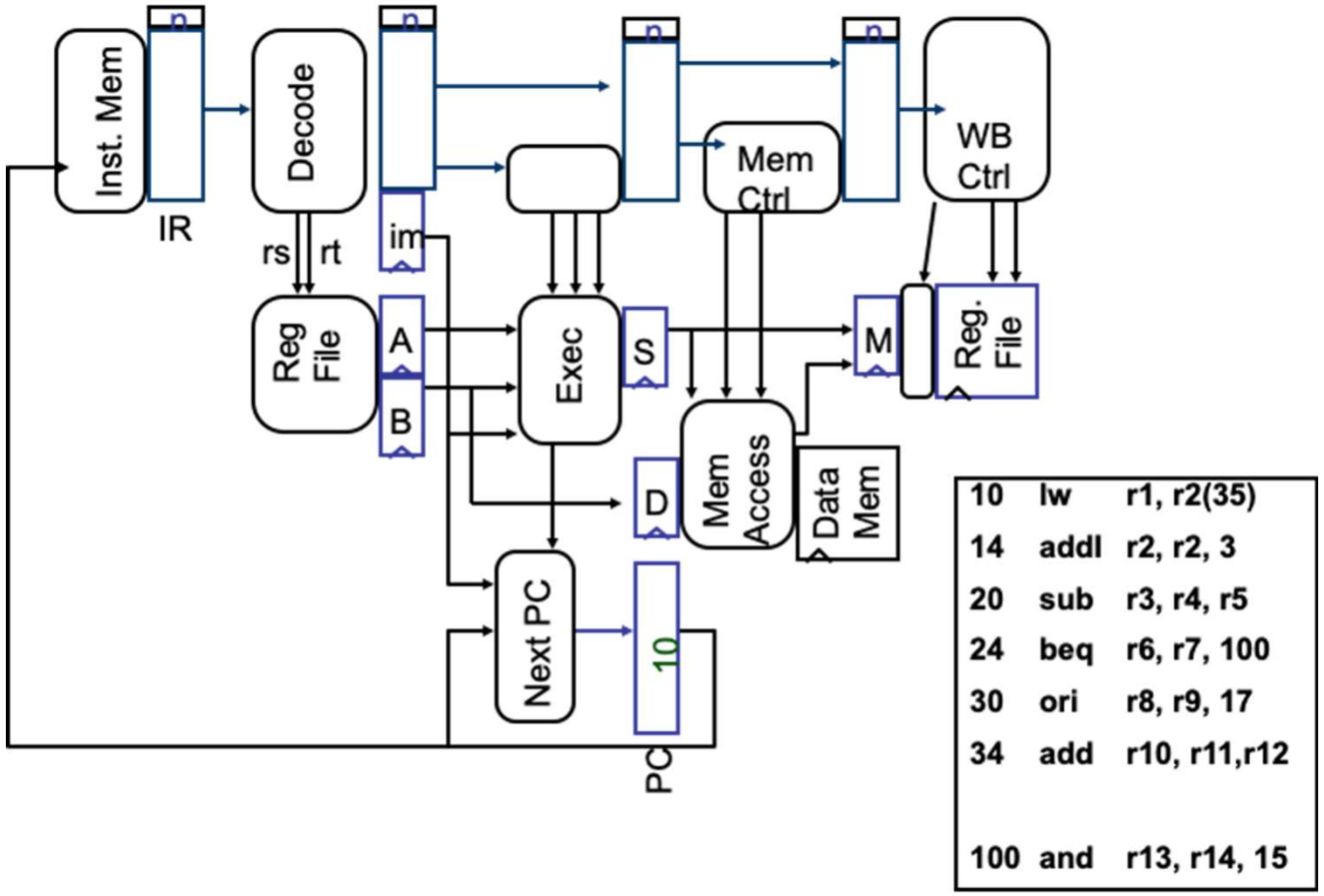


Give it a try

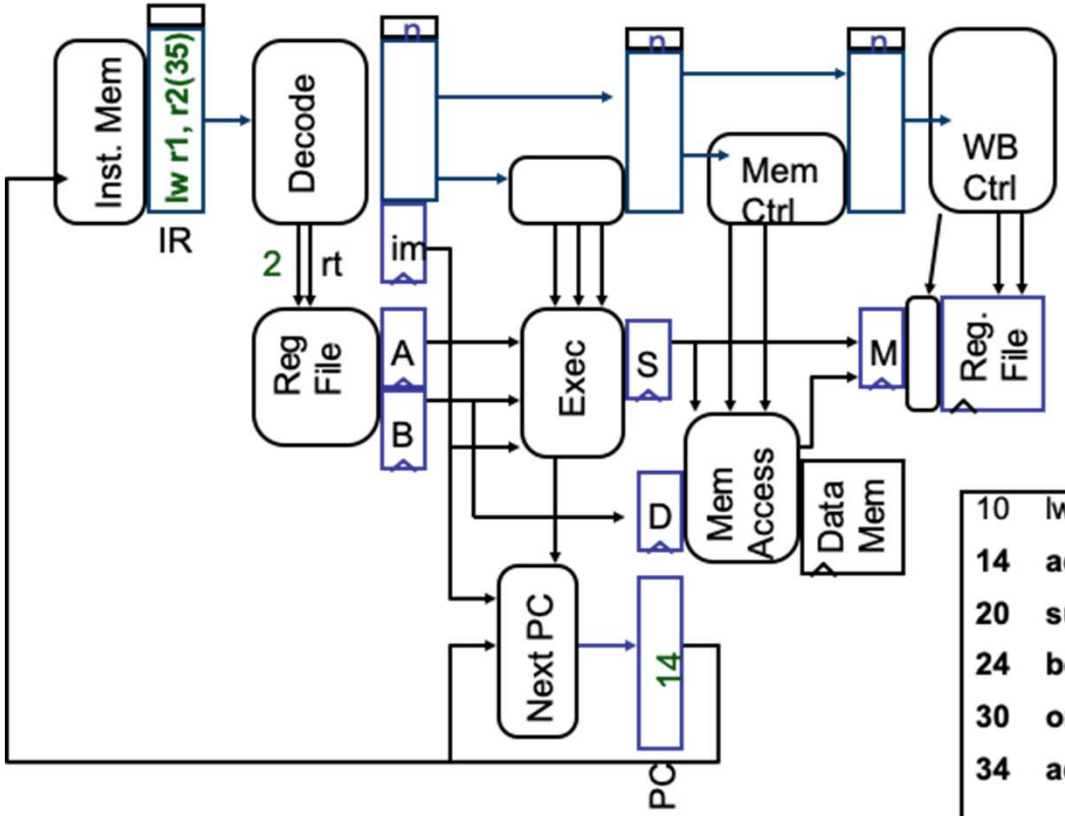
- ★ 10 lw r1, r2(35)
- ★ 14 addI r2, r2, 3
- ★ 20 sub r3, r4, r5
- ★ 24 beq r6, r7, 100
- ★ 30 ori r8, r9, 17
- ★ 34 add r10, r11, r12

- ★ 100 and r13, r14, 15

Fetch 10

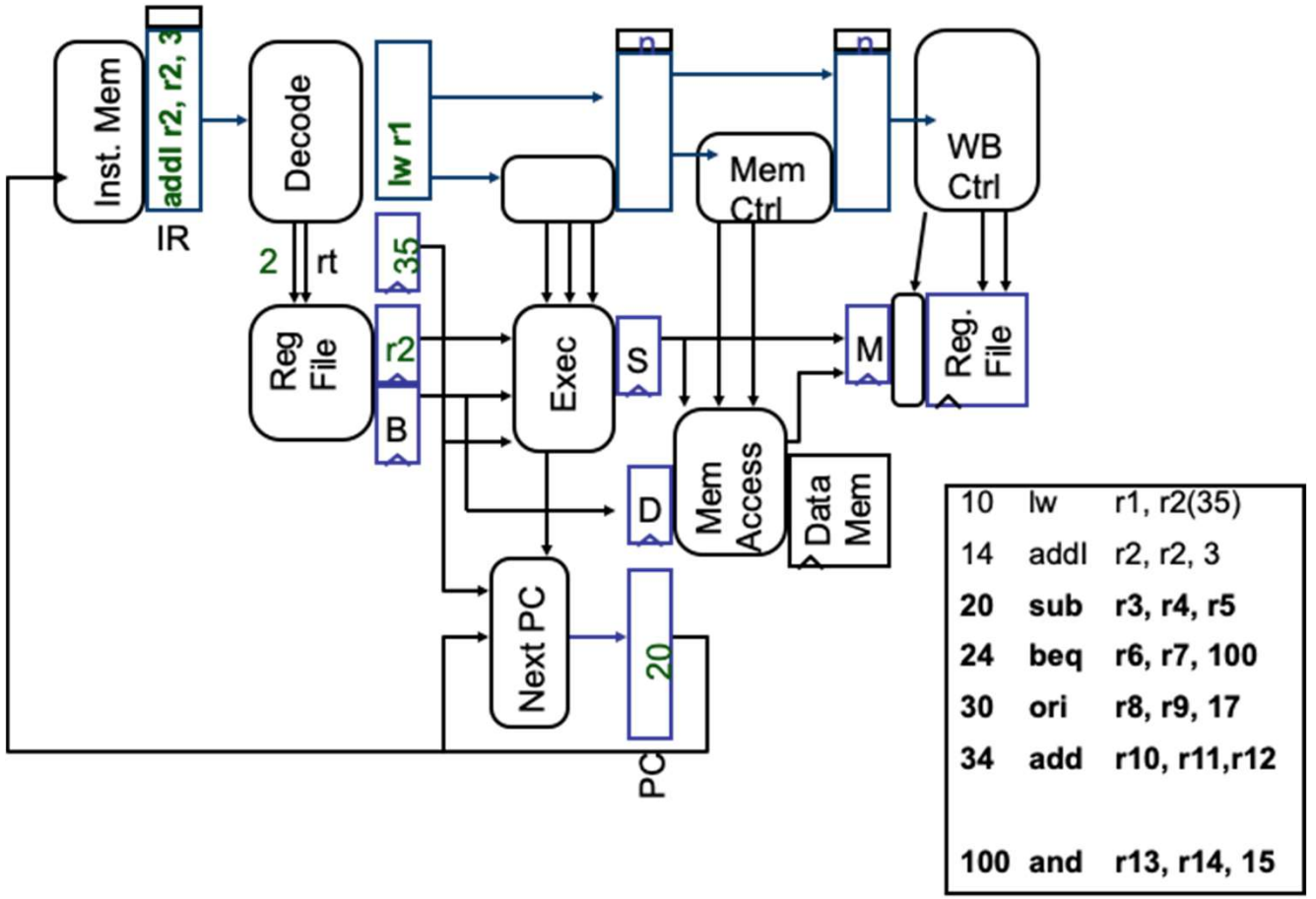

Inst. Mem
IR
Decode
rs
rt
im
Reg File
A
B
Exec
S
Mem Ctrl
WB Ctrl
M
Reg. File
Mem Access
Data Mem
D
Next PC
10
PC
10 lw r1, r2(35)
14 addl r2, r2, 3
20 sub r3, r4, r5
24 beq r6, r7, 100
30 ori r8, r9, 17
34 add r10, r11,r12
100 and r13, r14, 15

Fetch 14, Decode 10

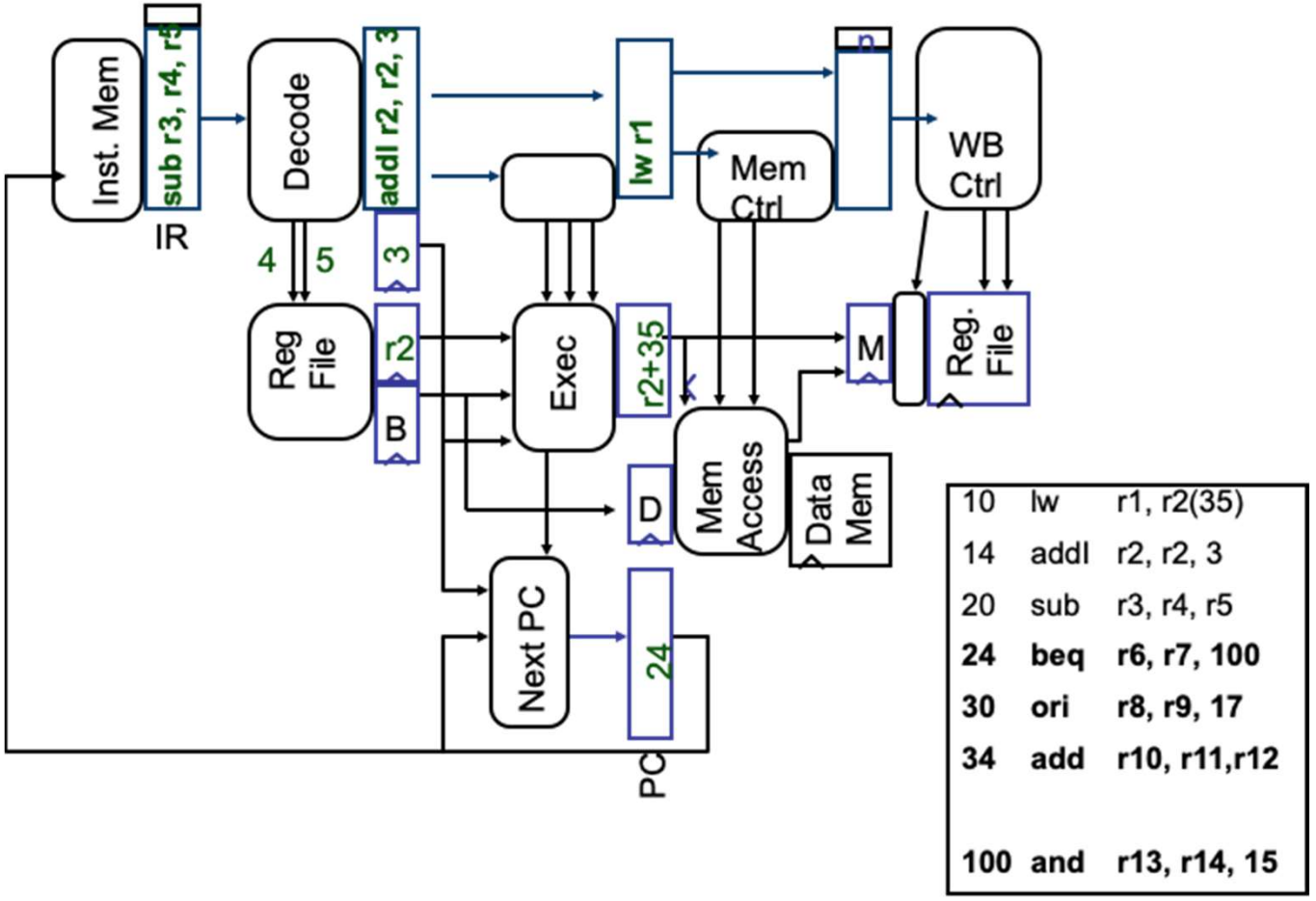


10	lw	r1, r2(35)
14	**addl**	**r2, r2, 3**
20	**sub**	**r3, r4, r5**
24	**beq**	**r6, r7, 100**
30	**ori**	**r8, r9, 17**
34	**add**	**r10, r11,r12**
100	**and**	**r13, r14, 15**

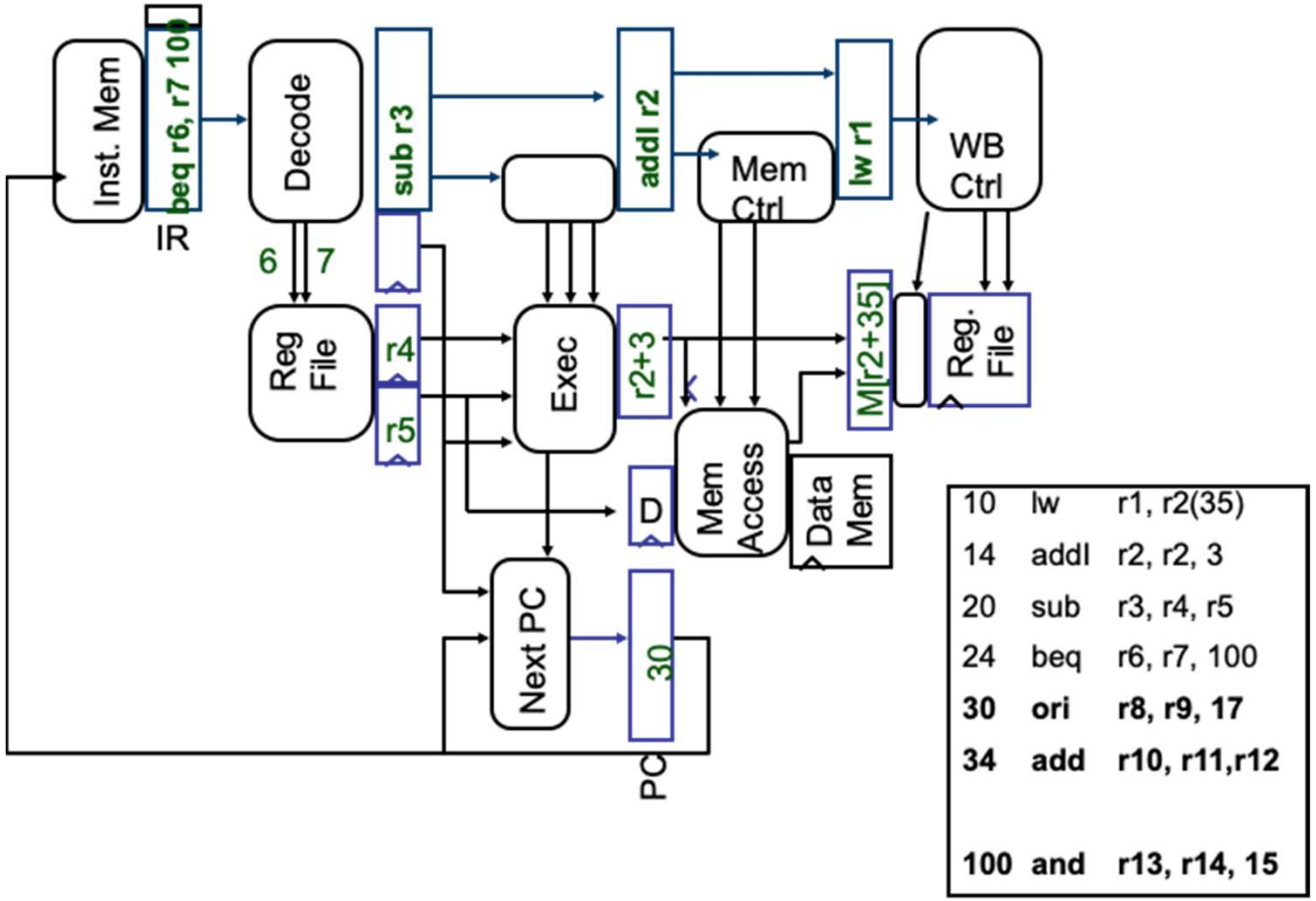
Fetch 20, Decode 14, Exec 10



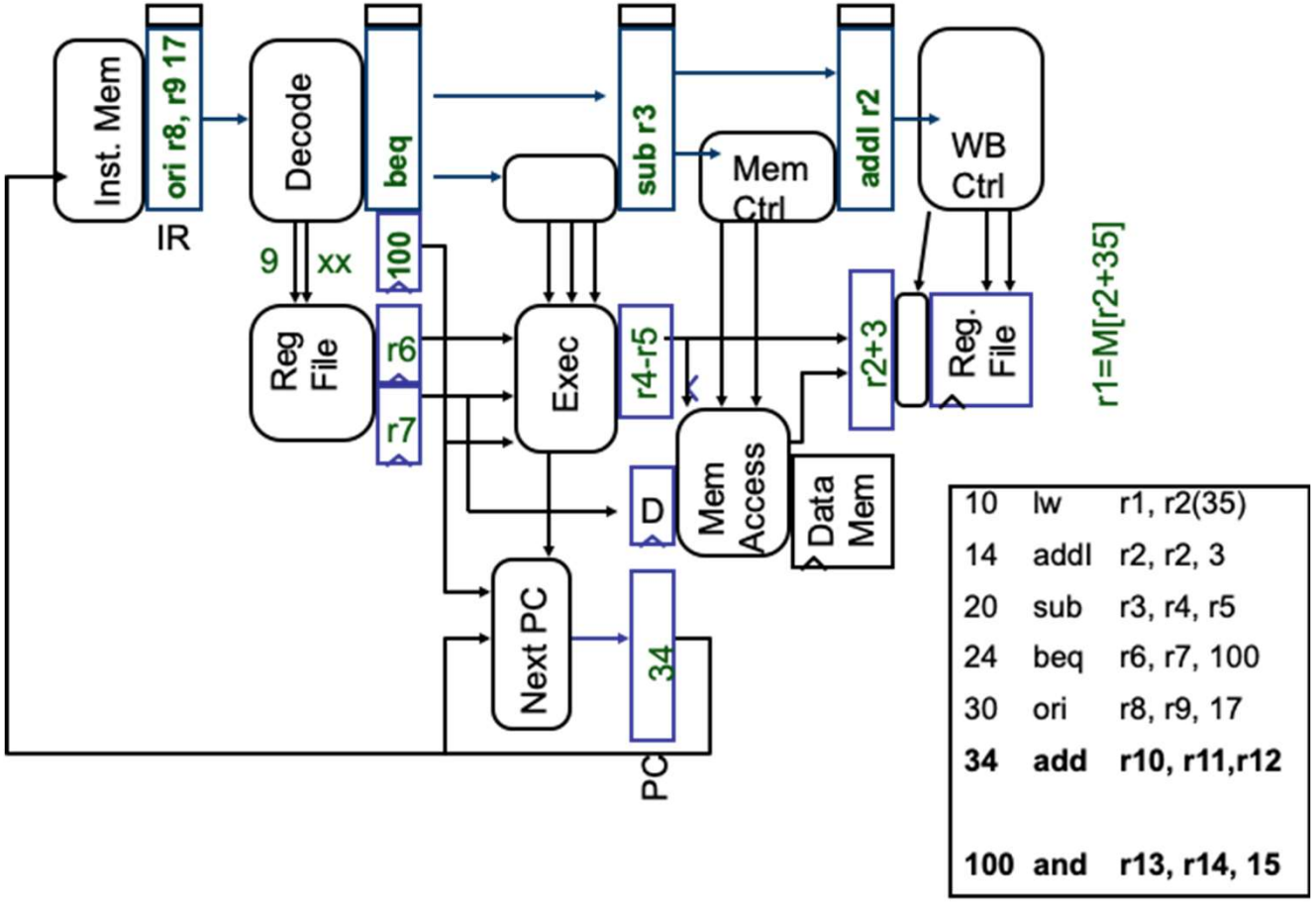
Fetch 24, Decode 20, Exec 14, Mem 10



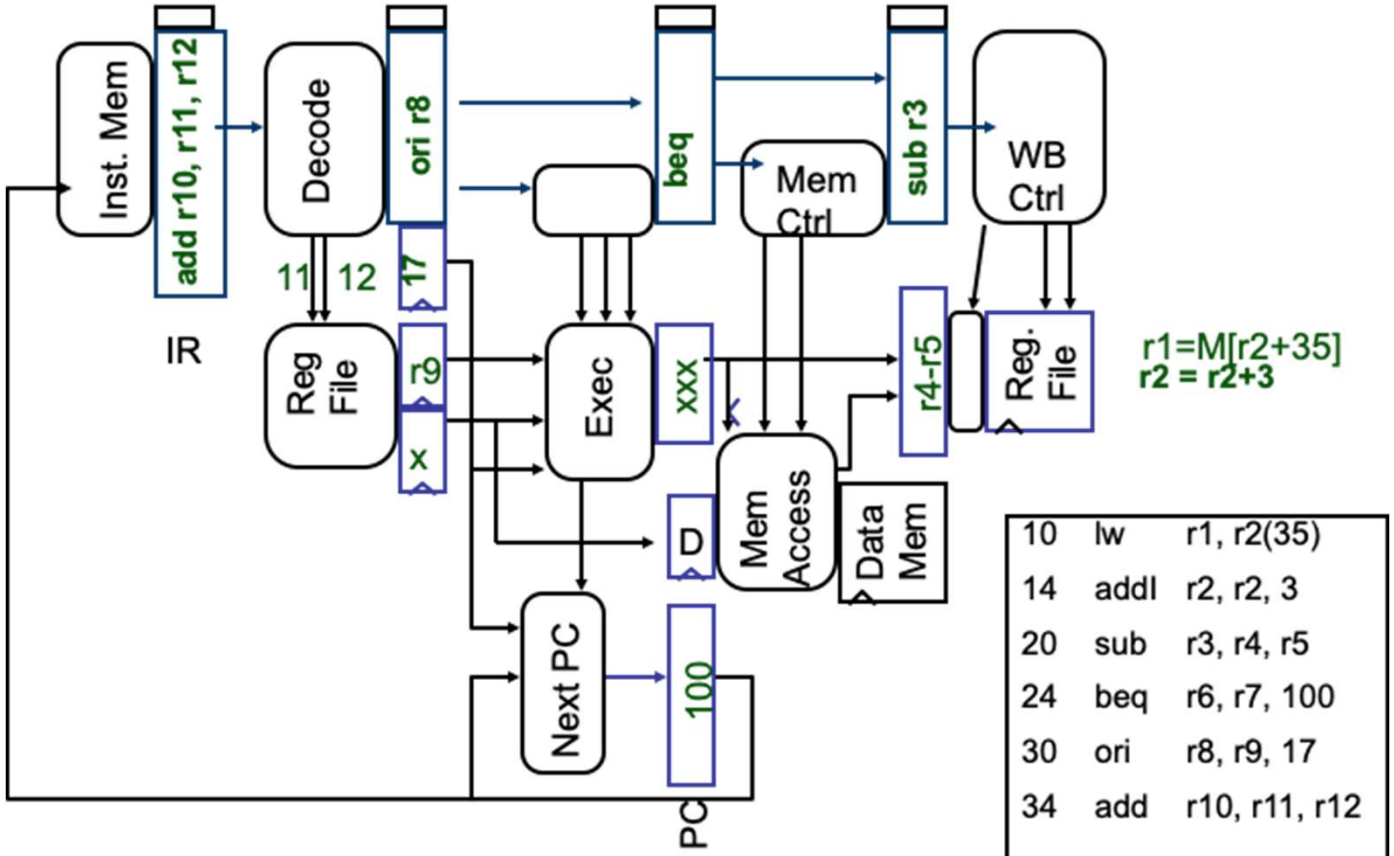
Fetch 30, Dcd 24, Ex 20, Mem 14, WB 10



Fetch 34, Dcd 30, Ex 24, Mem 20, WB 14



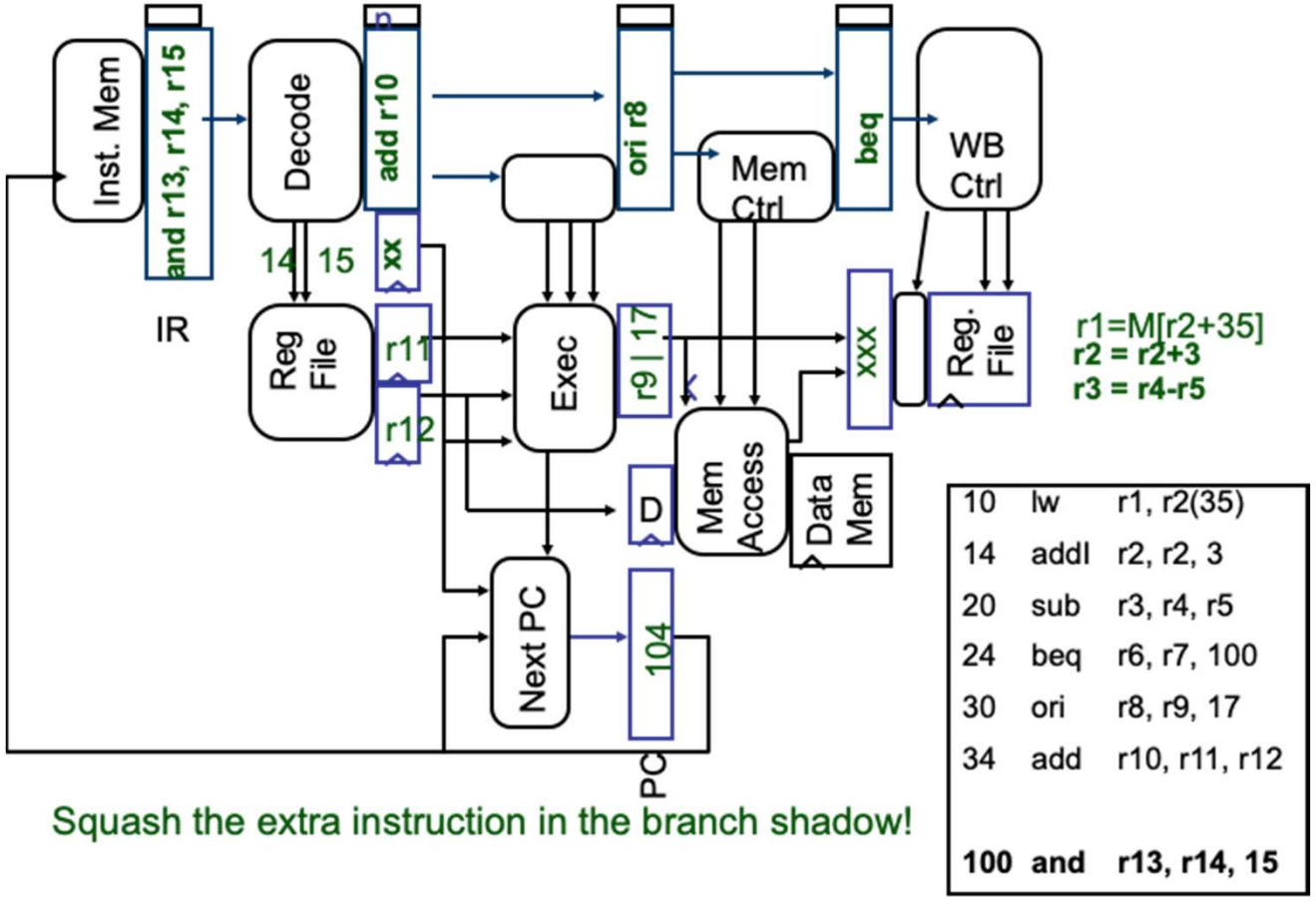
Fetch 100, Dcd 34, Ex 30, Mem 24, WB 20



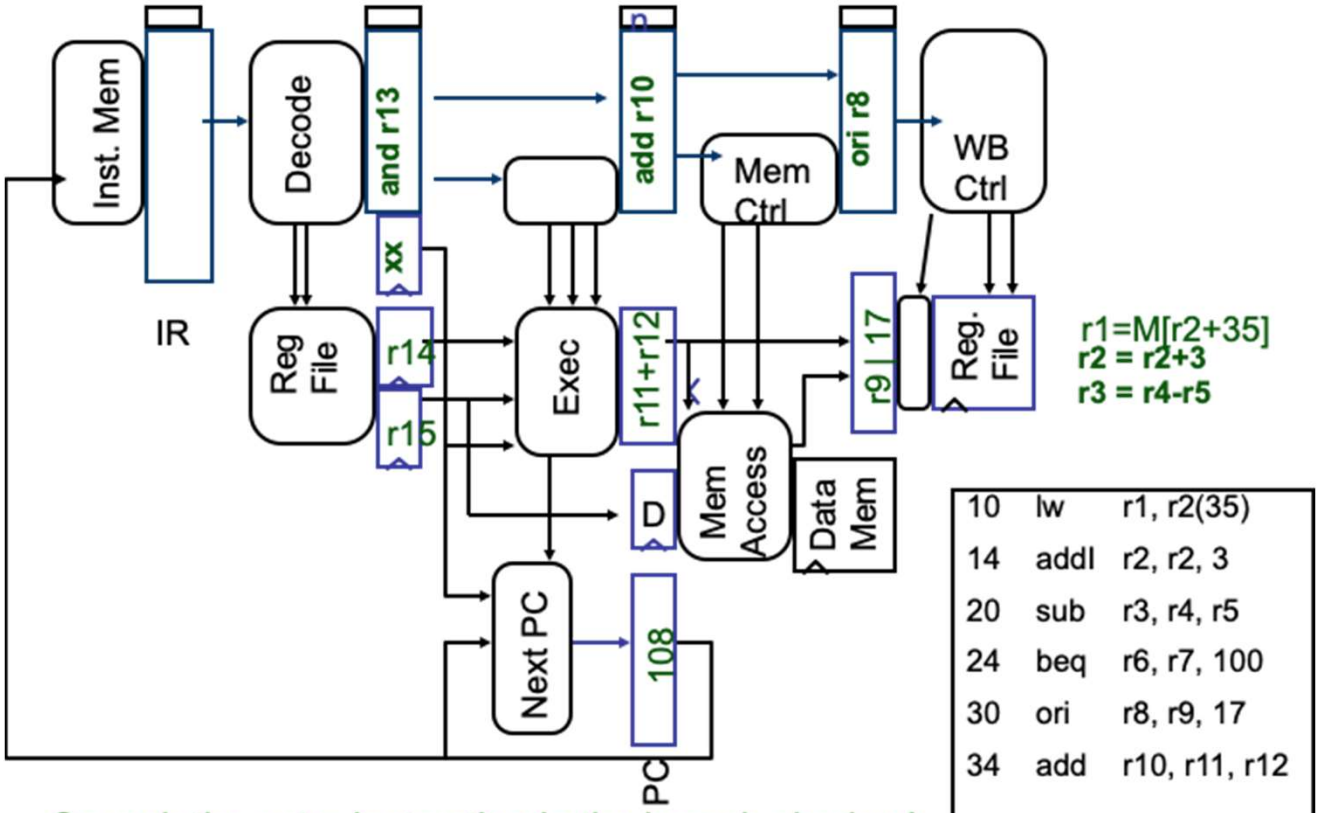

```
10   lw    r1, r2(35)
14   addI  r2, r2, 3
20   sub   r3, r4, r5
24   beq   r6, r7, 100
30   ori   r8, r9, 17
34   add   r10, r11, r12

100  and   r13, r14, 15
```

Fetch 104, Dcd 100, Ex 34, Mem 30, WB 24



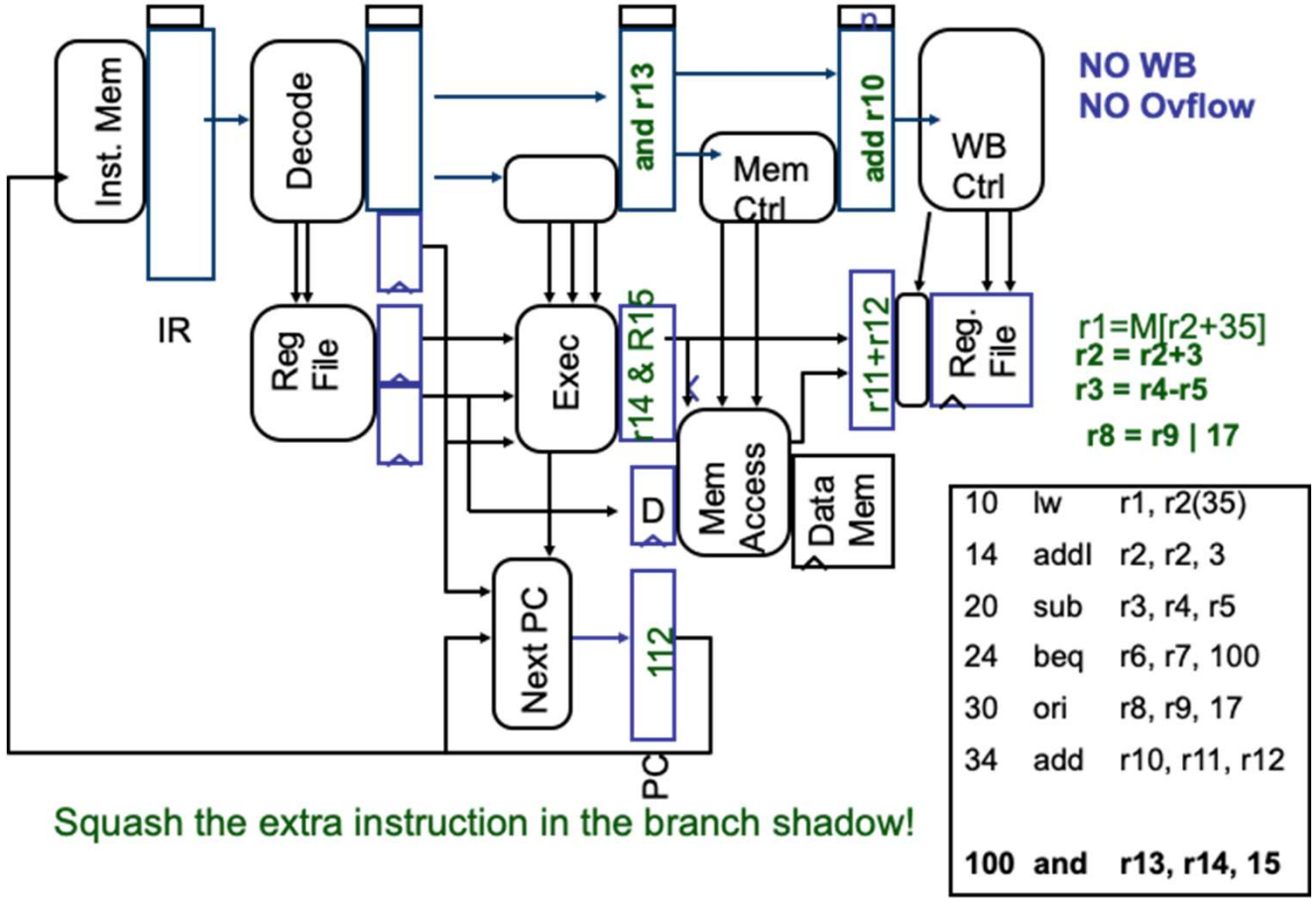
Fetch 108, Dcd 104, Ex 100, Mem 34, WB 30




```
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
```

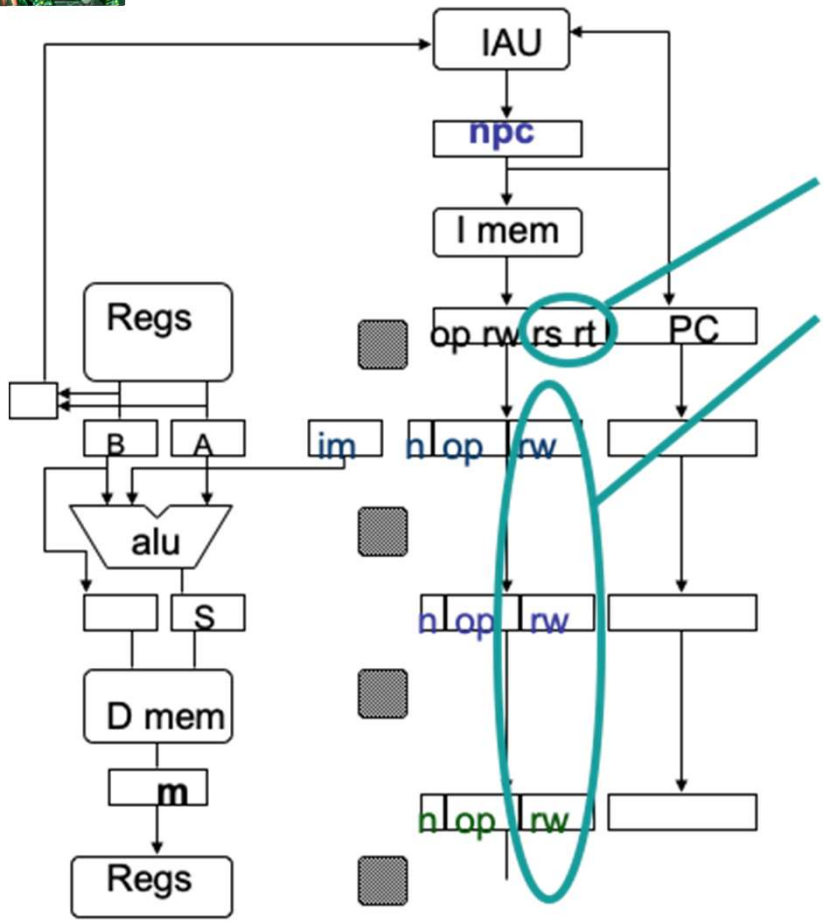
Fetch 112, Dcd 108, Ex 104, Mem 100, WB 34



Hazard Detection

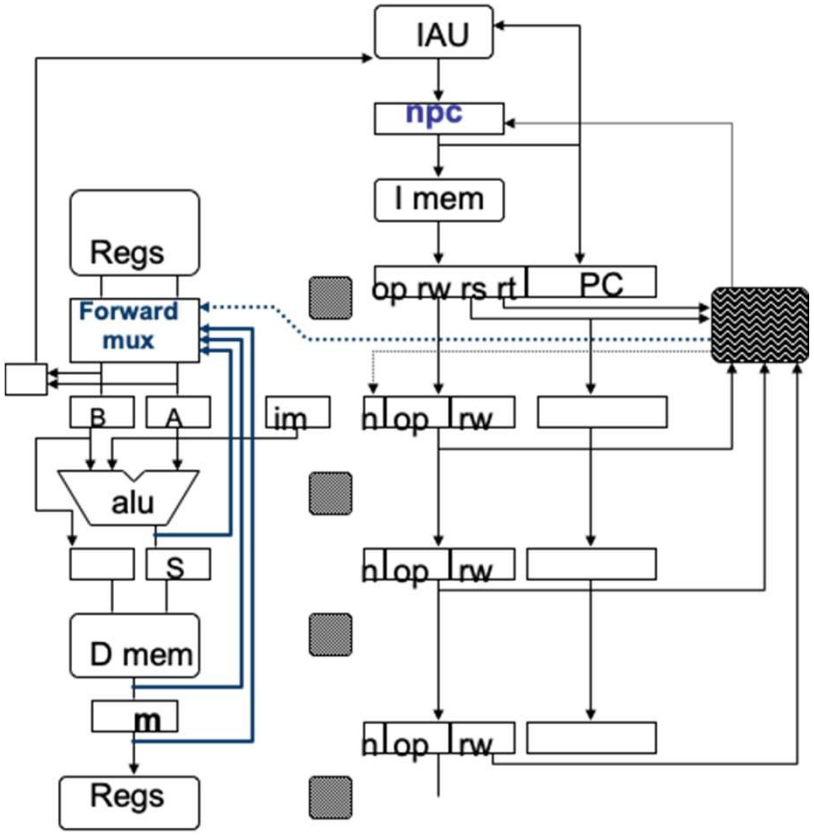
- ★ Suppose instruction i is about to be issued and a predecessor instruction j is in the instruction pipeline.
- ★ A RAW hazard exists on register r if r ∈ Rregs(i) ∩ Wregs(j)
  - Keep a record of pending writes (for inst's in the pipe) and compare with operand regs of current instruction.
  - When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

Hazard Detection Circuit



- Current operand registers
- Pending writes
- hazard <=
  - ((rs == rwex) & regWex) OR
  - ((rs == rwmem) & regWme) OR
  - ((rs == rwwb) & regWwb) OR
  - ((rt == rwex) & regWex) OR
  - ((rt == rwmem) & regWme) OR
  - ((rt == rwwb) & regWwb)

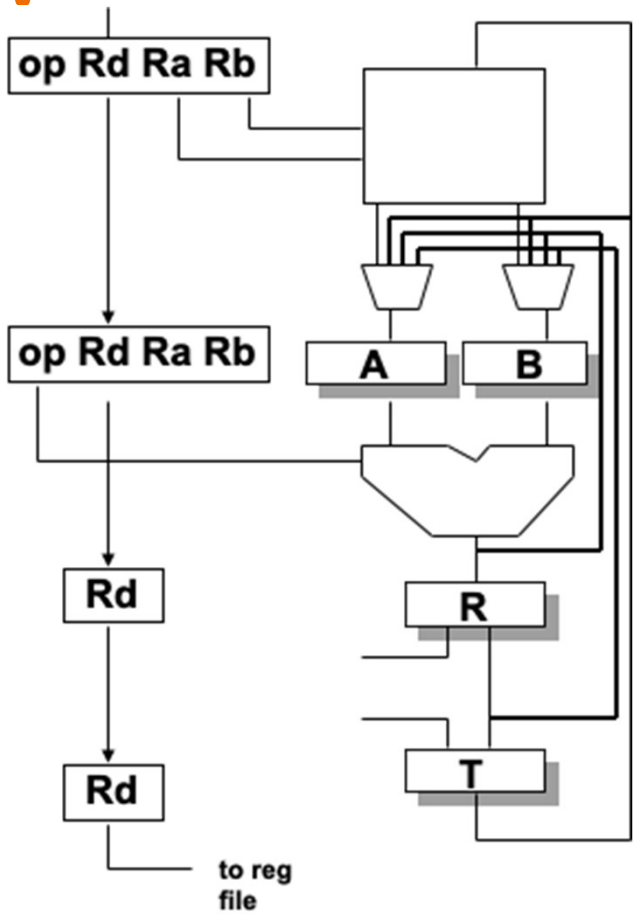
Forwarding Circuit



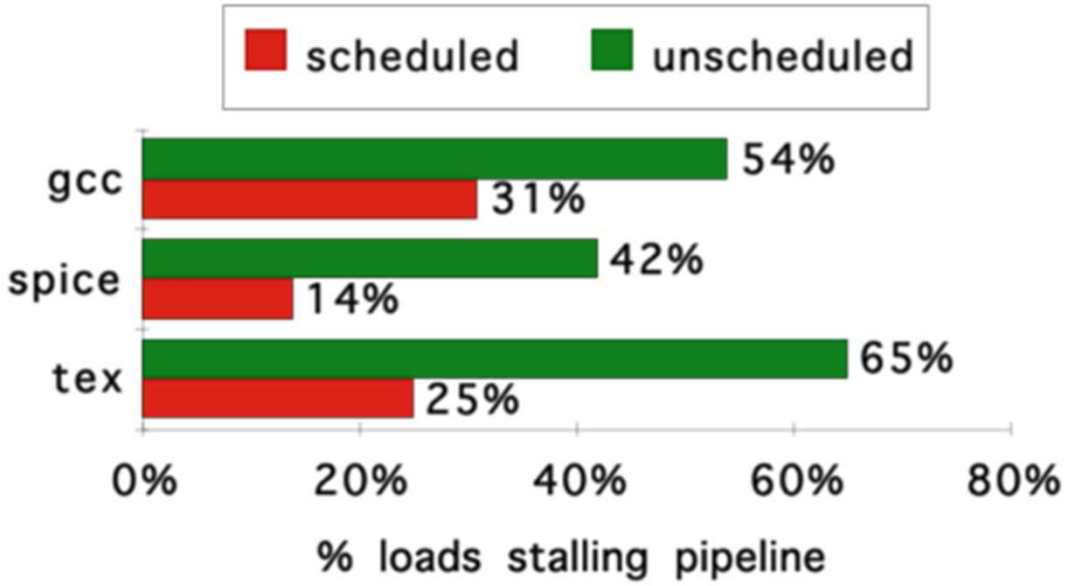
- ★ Detect nearest valid write op operand register and forward into op latches, bypassing remainder of the pipe
- ★ Increase muxes to add paths from pipeline registers
- ★ Data Forwarding = Data Bypassing

What about memory operations?

- If instructions are initiated in order and operations always occur in the same stage, there can be no hazards between memory operations!
- What does delaying WB on arithmetic operations cost?
  - cycles ?
  - hardware ?
- What about data dependence on loads?
  - R1 <- R4 + R5
  - R2 <- Mem[R2 + I]
  - R3 <- R2 + R1
- => “Delayed Loads”



Compiler scheduling to avoid load stalls



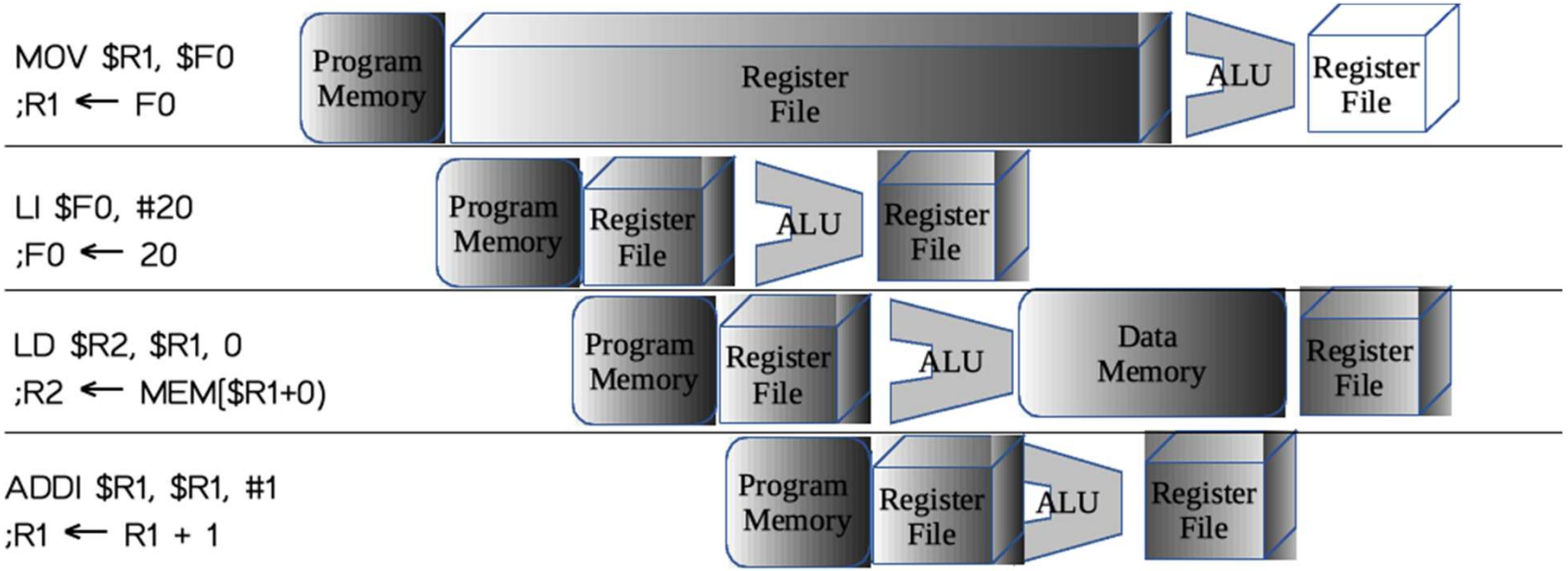
Summary

- ★ Pipelining is a fundamental concept
- ★ multiple steps using distinct resources
- ★ Utilize capabilities of the Datapath by pipelined instruction processing
- ★ start next instruction while working on the current one
- ★ limited by length of longest stage (plus fill/flush)
- ★ detect and resolve hazards

Exercises

Data Hazard

★ Identify all data hazards in this diagram. (WAR, WAW, RAW)



Rescheduling

- Assuming that the hardware has a full support for forwarding hardware, please reschedule the following code to take the benefit of delayed branch and delayed load. You answer must be free from stall cycle (if possible)
- Loop: LD $F0, 0($R1)
  - ADD $F4, $F0, $F2
  - SD 0($R1), $F4
  - LD $F1, 8($R1)
  - ADD $F5, $F1, $F2
  - SD 8($R1), $F5
  - SUB $R1, $R1, #16
  - BNEZ $R1, Loop

End of Chapter 6 (part 2)