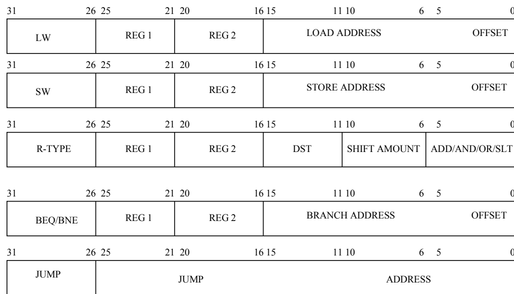
Instruction Format



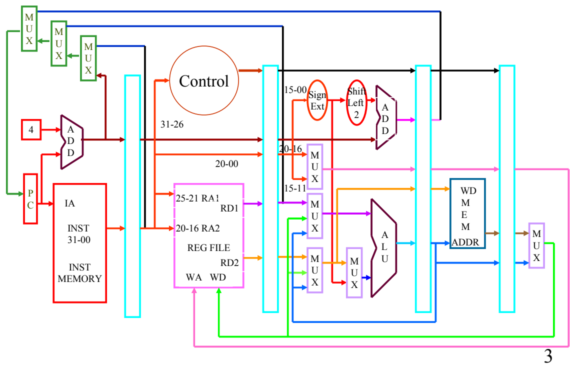
1

Operation for Each Instruction

LW:	SW:	R-Type:	BR-Type:	JMP-Type:
1. READ INST	1. READ INST	1. READ INST	1. READ INST	1. READ INST
2. READ REG 1 *READ REG 2*	2. READ REG 1 READ REG 2	2. READ REG 1 READ REG 2	2. READ REG 1 READ REG 2	2.
3. ADD REG 1 + OFFSET	3. ADD REG 1 + OFFSET	3. OPERATE on REG 1 / REG 2	3. SUB REG 2 from REG 1	3.
4. READ MEM	4. WRITE MEM	4.	4.	4.
5. WRITE REG2	5.	5. WRITE DST	5.	5.

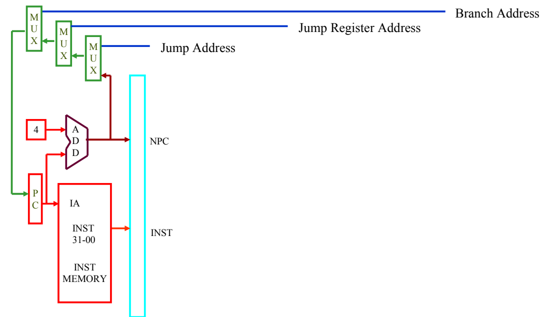
2

Pipeline Data Path Operation



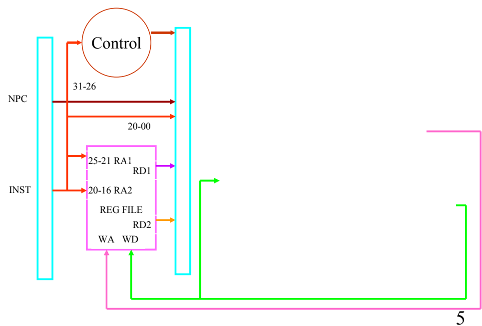
3

Fetch Unit



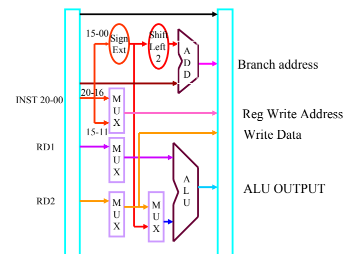
4

Register Fetch Unit



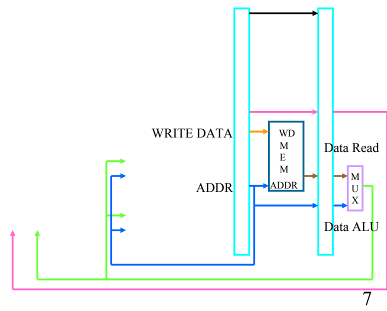
5

ALU Operation and Branch Logic

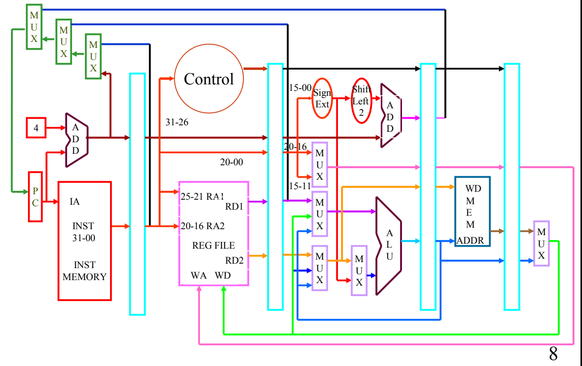


6

Memory and Write back Stage



Pipeline Data Path Operation



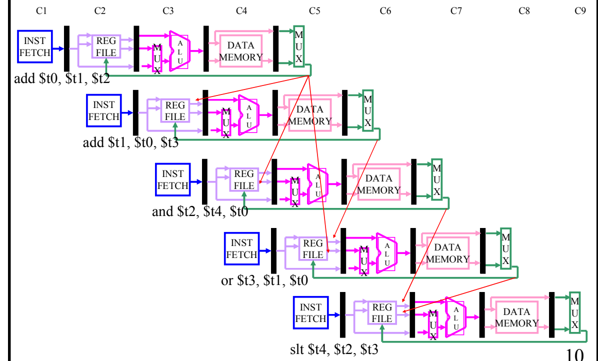
A program with data dependencies

- Consider the following program

```
add  $t0, $t1, $t2
add  $t1, $t0, $t3
and  $t2, $t4, $t0
or   $t3, $t1, $t0
slt  $t4, $t2, $t3
```

- Problem with starting next instruction before first is finished
  - dependencies that "go backward in time" are data hazards

Data Path Operation



Solution: Software No-ops/Hardware Bubbles

- Have compiler guarantee no hazards
- Where do we insert the "no-ops" ?

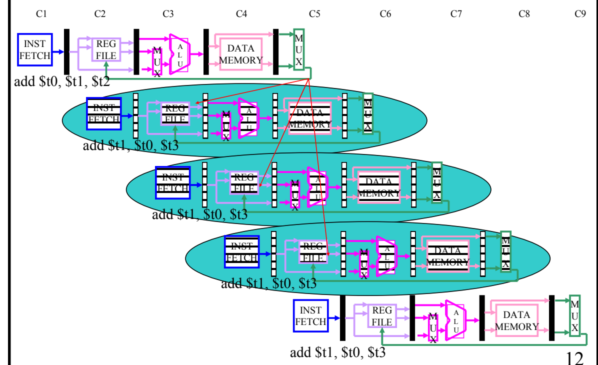
```
sub   $2, $1, $3
and   $12, $2, $5
or    $13, $6, $2
add   $14, $2, $2
sw    $15, 100($2)
```

Problem: this really slows us down!

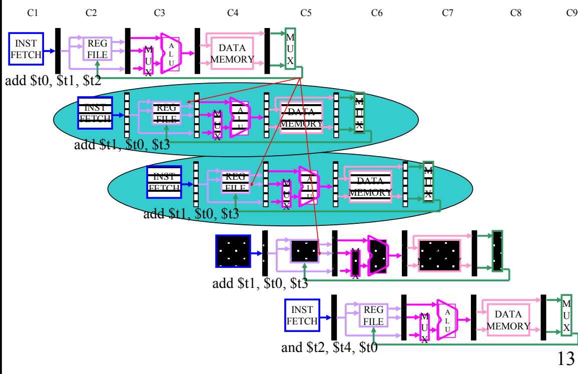
- Also, the program will always be slow even if a techniques like forwarding is employed afterwards in newer version

- Hardware can detect dependencies and insert no-ops in hardware
  - Hardware detection and no-op insertion is called stalling
  - This is a bubble in pipeline and waste one cycle at all stages
  - Need two or three bubbles between write and read of a register

Stalled Operation (no write before read)

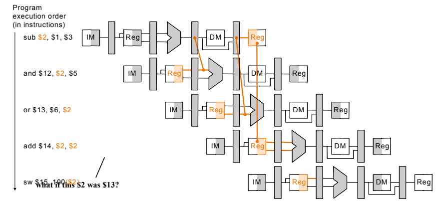


Stalled Operation (write before read)



Forwarding

- Use temporary results, don't wait for them to be written
  - register file forwarding to handle read/write to same register
  - ALU forwarding
  - May also need forwarding to memory (think!!)

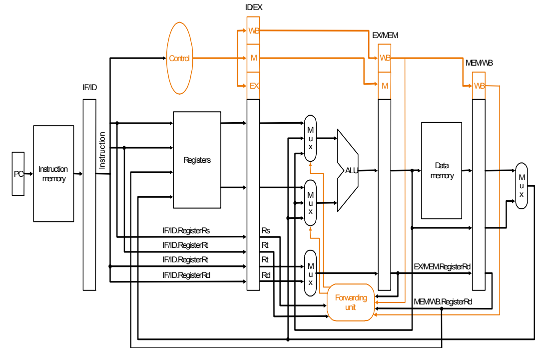


Detecting Hazards for Forwarding

- **EX hazard**
  - If ((EX/MEM.RegWrite) and (EX/MEM.RegisterRd != 0) and (EX/MEM.REgisterRd = ID/EX.RegisterRs)) ForwardA = 10
  - If ((EX/MEM.RegWrite) and (EX/MEM.RegisterRd != 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRt)) ForwardB = 10
- **MEM hazard**
  - If ((MEM/WB.RegWrite) and (MEM/WB.REgisterRd != 0) and (MEM/WB.REgisterRd = ID/EX.RegisterRs)) ForwardA = 01
  - If ((MEM/WB.RegWrite) and (MEM/WB.REgisterRd != 0) and (MEM/WB.REgisterRd = ID/EX.RegisterRt)) ForwardB = 10

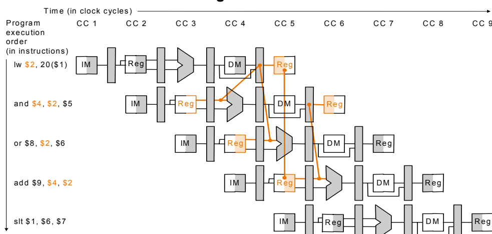
- **In case of lw followed by a sw instruction, forwarding will not work. This is because data in MEM stage are still being read**
  - **Plan on adding forwarding in MEM stage of put a stall/bubble**
- **In case of lw followed by an instruction that uses the value**
  - **One has to add an stall**

Forwarding



Can't always forward

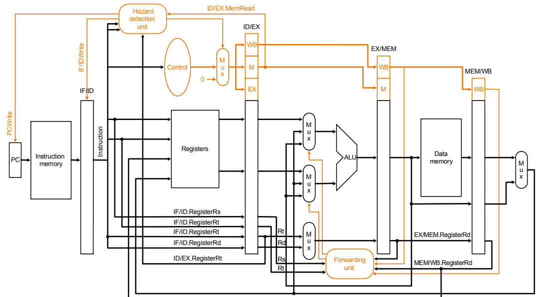
- Load word can still cause a hazard:
  - an instruction tries to read a register following a load instruction that writes to the same register.



- Thus, we need a hazard detection unit to "stall" the load instruction

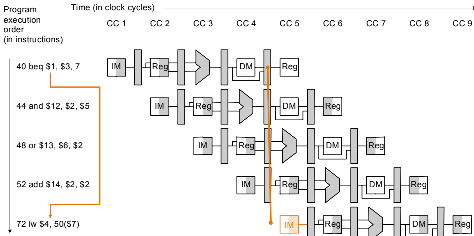
Hazard Detection Unit

- Stall by letting an instruction that won't write anything go forward



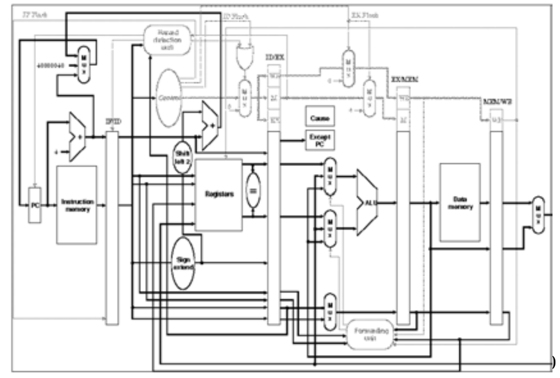
Branch Hazards

- When we decide to branch, other instructions are in the pipeline!



- We are predicting "branch not taken"
  - need to add hardware for flushing instructions if we are wrong

Flushing Instructions



Improving Performance

- Try and avoid stalls! E.g., reorder these instructions:

```
lw $t0, 0($t1)
lw $t2, 4($t1)
sw $t2, 0($t1)
sw $t0, 4($t1)
```

- Add a "branch delay slot"
  - the next instruction after a branch is always executed
  - rely on compiler to "fill" the slot with something useful
- Superscalar: start more than one instruction in the same cycle

Other Issues in Pipelines

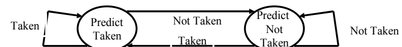
- Exceptions
  - Errors in ALU for arithmetic instructions
  - Memory non-availability
- Exceptions lead to a jump in a program
- However, the current PC value must be saved so that the program can return to it back for recoverable errors
- Multiple exception can occur in a pipeline
- Preciseness of exception location is important in some cases
- I/O exceptions are handled in the same manner

Handling Branches

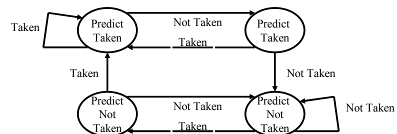
- Branch Prediction
  - Usually we may simply assume that branch is not taken
  - If it is taken, then we flush the pipeline
    - Clear control signals for instruction following branch
- Delayed branch
  - Fill instructions that need to be executed even if branch occur
  - If none available fill NOOPs
- Reduce delay in resolving branches
  - Compare at register stage
  - Branch prediction table
    - PC value (for branch) and next address
    - One or two bits to store what should be prediction

Two State vs Four State Branch Prediction

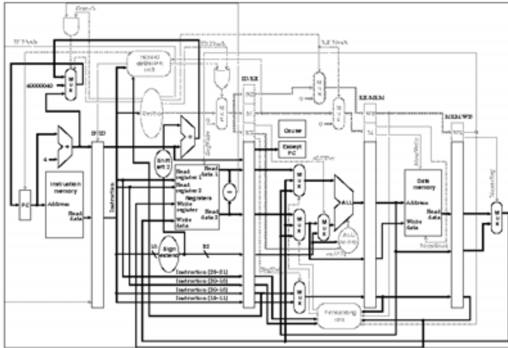
- Two state model



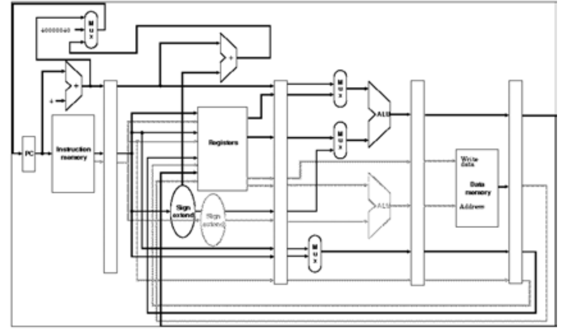
- Four State Model



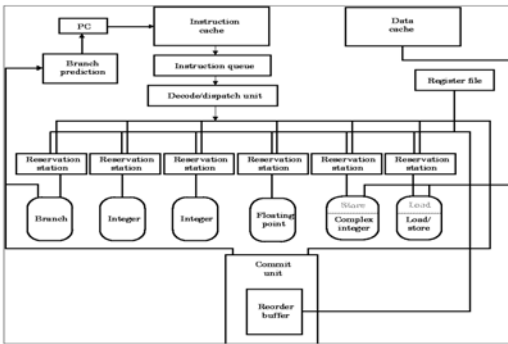
Pipeline with Early Branch Resolution/Exception



Superscalar Architecture



A Modern Pipelined Microprocessor



Important Facts to Remember

- Pipelined processors divide the execution in multiple steps
- However pipeline hazards reduce performance
  - Structural, data, and control hazard
- Data forwarding helps resolve data hazards
  - But all hazards cannot be resolved
  - Some data hazards require bubble or noop insertion
- Effects of control hazard reduced by branch prediction
  - Predict always taken, delayed slots, branch prediction table
  - Structural hazards are resolved by duplicating resources

Execution Time

- Time of n instructions depends on
  - Number of instructions n
  - # of stages k
  - # of control hazard and penalty of each step
  - # of data hazards and penalty for each
- Time = n + k - 1 + load hazard penalty + branch penalty
- Load hazard penalty is 1 or 0 cycle
  - depending on data use with forwarding
- branch penalty is 3, 2, 1, or zero cycles depending on scheme

Design and Performance Issues With Pipelining

- Pipelined processors are not EASY to design
- Technology affect implementation
- Instruction set design affect the performance, i.e., beq, bne
- More stages do not lead to higher performance