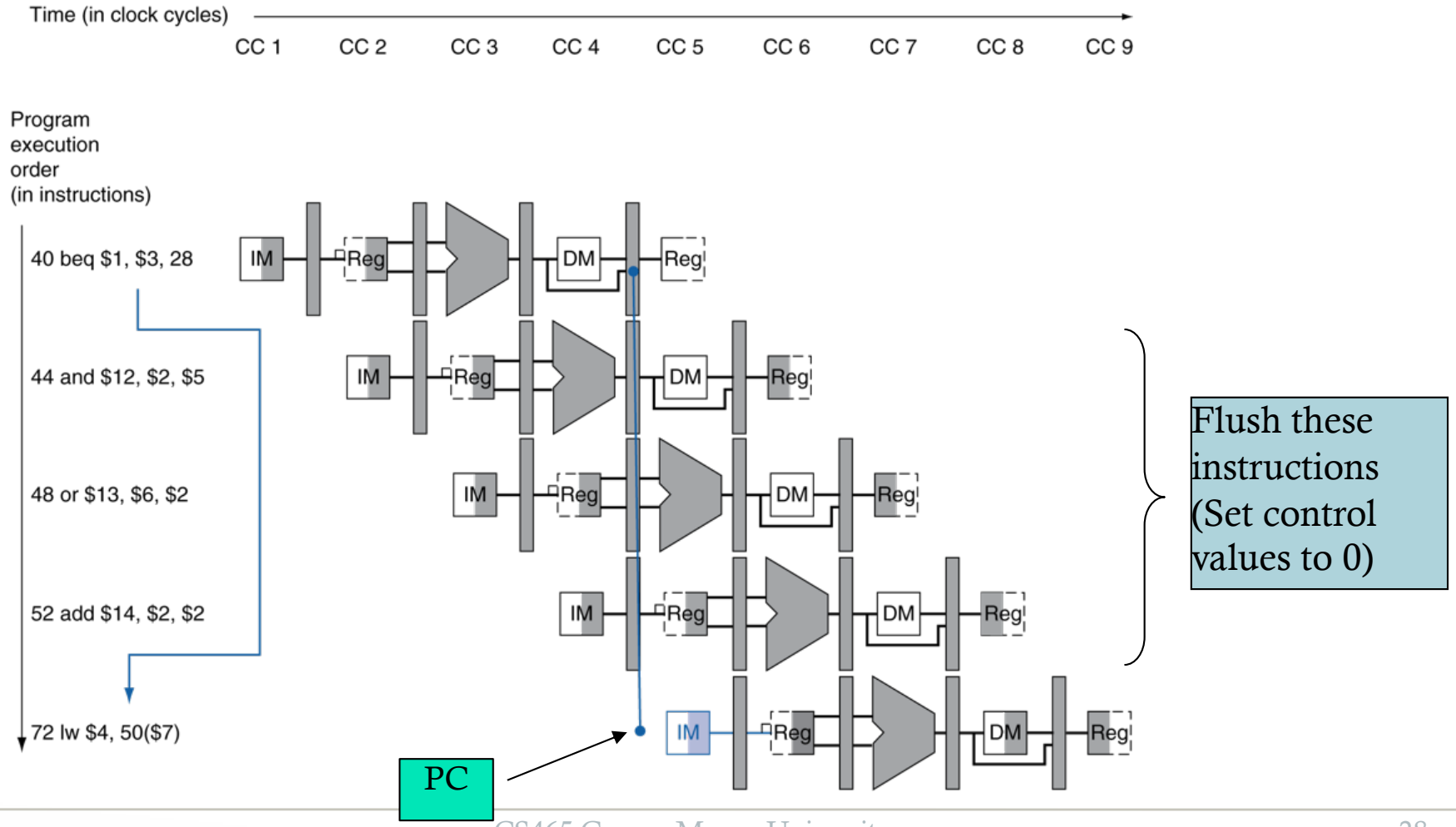
Roadmap

- Resolving hazards
  - Data hazards: forwarding
  - Stalling the pipeline
  - Control hazards; prediction
- Advanced ILP techniques
  - Reordering
  - Static multiple issue
  - Dynamic scheduling and speculation
- Concluding remarks

Control Hazards

- If branch outcome determined in MEM stage:



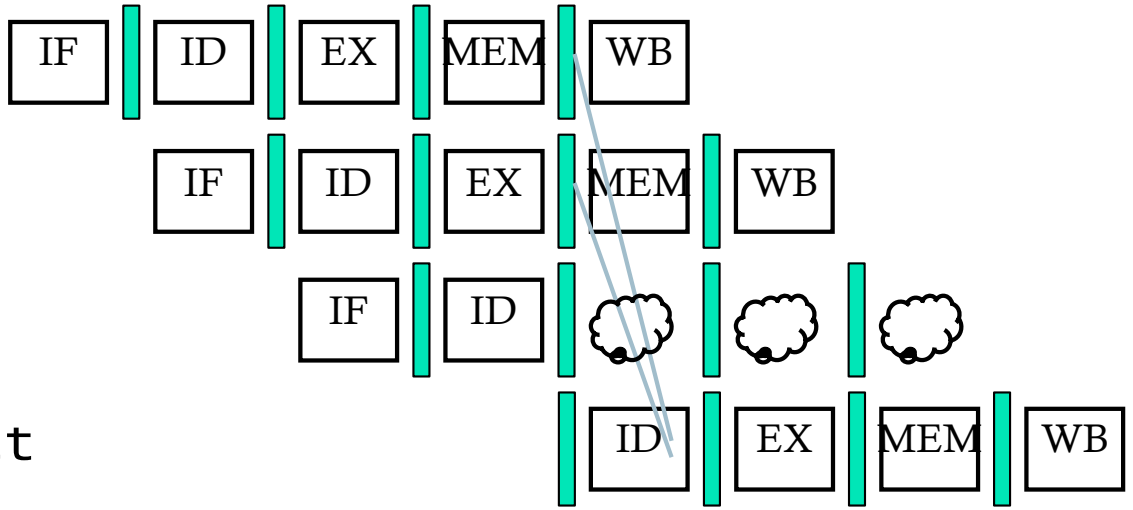
Observations

- Basic implementation
  - Branch decision does not occur until MEM stage
  - 3 CCs are wasted
- How to decide branch earlier and reduce delay
  - In EX stage - two CCs branch delay
  - In ID stage - one CC branch delay
  - How?
    - For beq $x, $y, label, $x xor $y then or all bits, much faster than ALU operation
    - Also we have a separate ALU to compute branch address
    - May need additional forwarding and suffer from data hazards

Data Hazards for Branches

- If a comparison register is a destination of preceding ALU instruction or 2nd preceding load instruction
  - Need 1 stall cycle

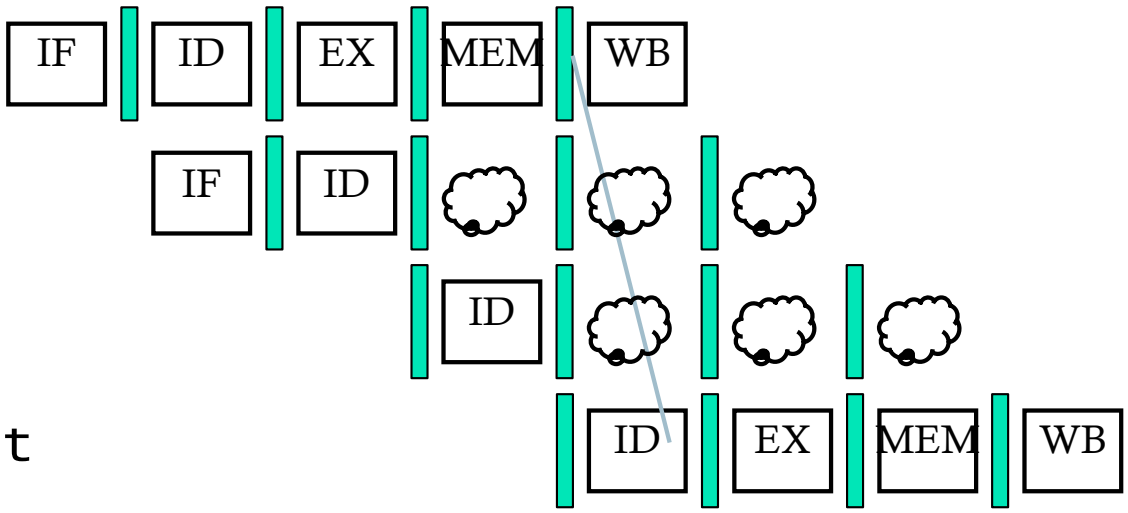
```
lw   $1, addr
add $4, $5, $6
beq stalled
beq $1, $4, target
```



Data Hazards for Branches

- If a comparison register is a destination of immediately preceding load instruction
  - Need 2 stall cycles

```
lw  $1, addr
beq stalled
beq stalled
beq $1, $0, target
```

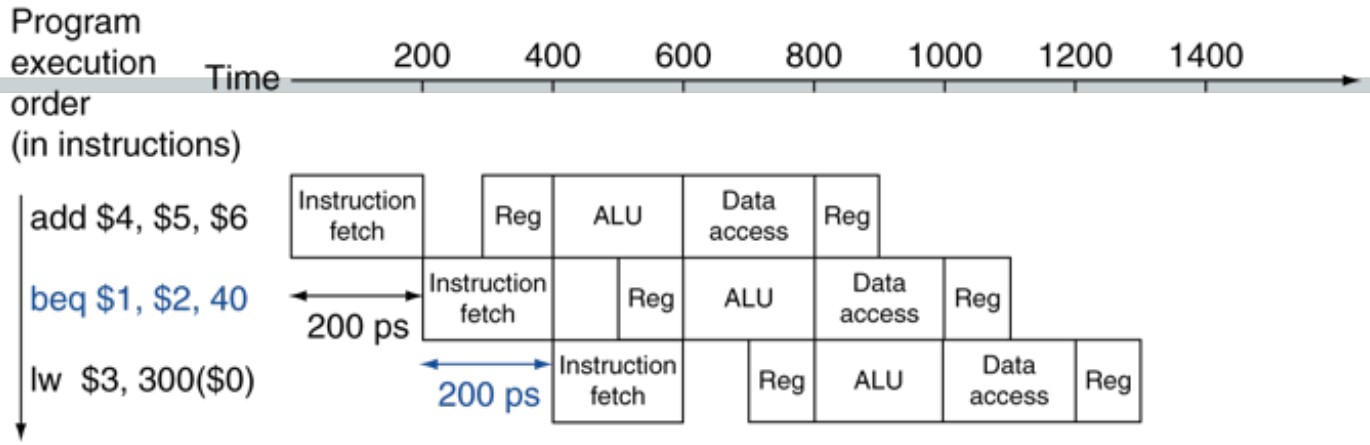


Branch Prediction

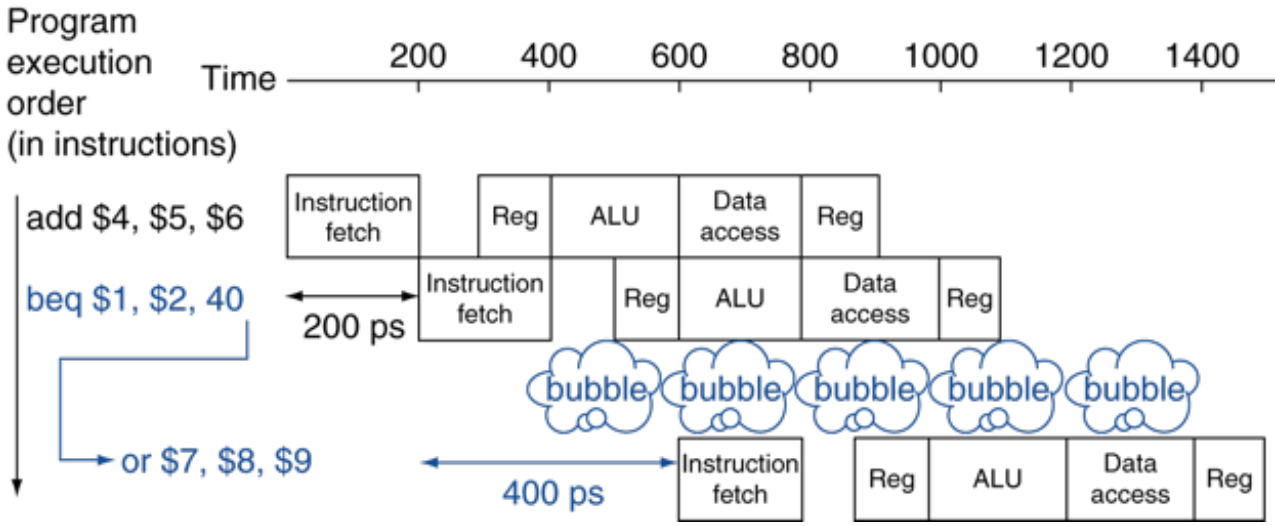
- Assume that branch decision is made at ID stage
  - → 1 stall cycle
- Solution: predict outcome of branch and proceed
  - If guess wrong, abort and restart from the right location
- Simple predictions
  - Easier to predict branches as "always not taken"
    - Fetch instruction after branch, with no delay
    - No stall if guess correct!
    - No additional delay if guess wrong!
  - Can also predict as "always taken"
    - Need to figure out the branch target

Proceed as "Not taken"

Prediction correct



Prediction incorrect



History-based Branch Prediction

- Always taken/not-taken branch prediction is crude!
  - Different branches/programs vary a lot
  - Better: take history into consideration
- Branch history table / branch prediction buffer
  - One entry for each branch, containing a bit (or bits) which tells whether the branch was recently taken or not
    - Indexed by the lower bits of the branch instruction address
    - Table lookup might occur in stage IF
  - How many bits for each table entry?
  - What to do if the prediction incorrect?

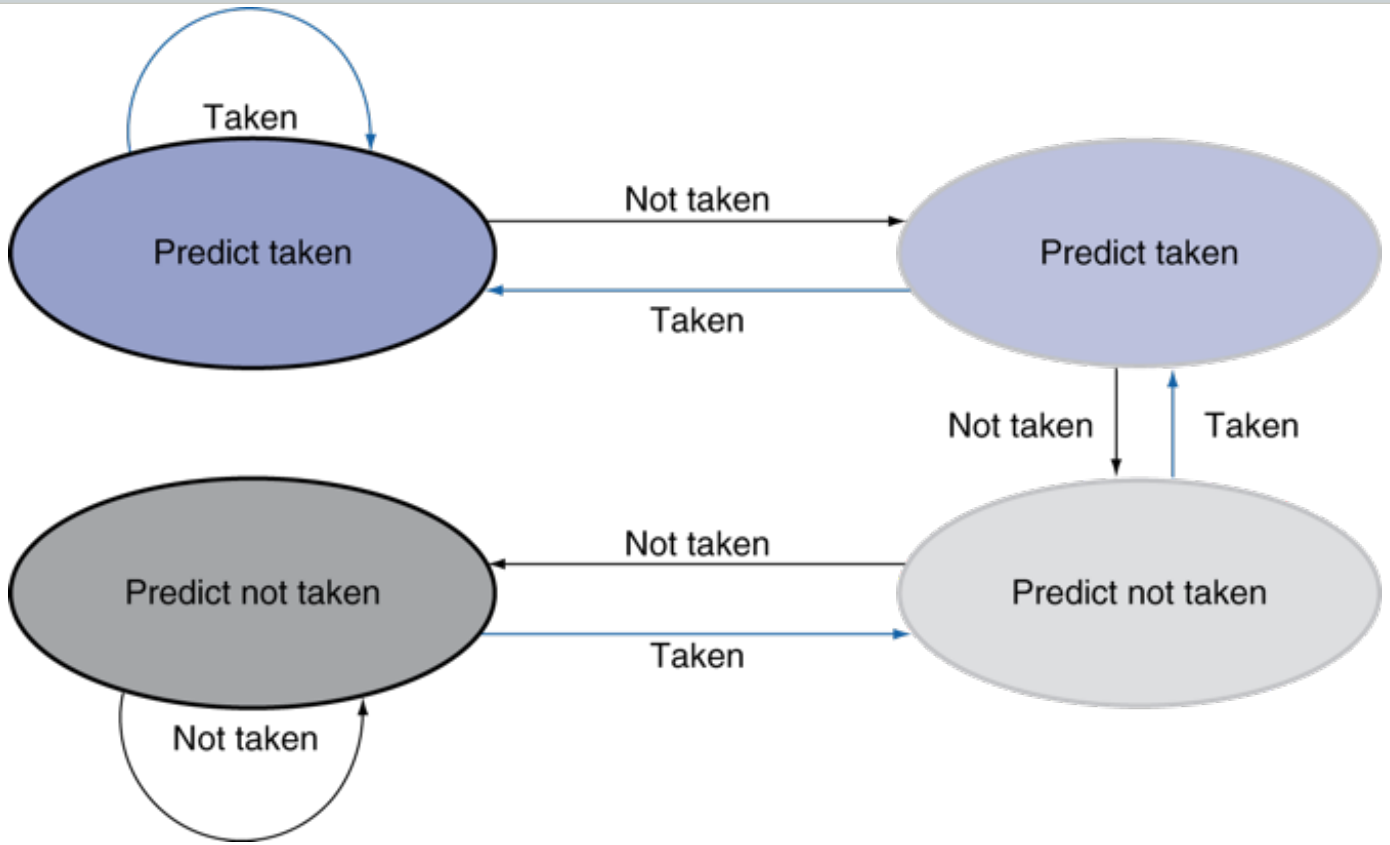
Dynamic Branch Prediction

- Simplest approach: 1-bit prediction
  - Record whether or not branch taken last time
  - Always predict branch will behave the same as last time
  - Mis-prediction will cause the single prediction bit flipped
- Problem: even if a branch is almost always taken, we will likely predict incorrectly twice
  - Consider a loop: T, T, …, T, NT, T, T, …

Exercise

- Consider a loop branch (inner loop) that is taken nine times in a row, then is not taken once. What is the prediction accuracy for this branch?
  - Assume 1-bit history
  - Assume we initialize to predict taken

2-bit Predictor



- Only change prediction if history shows strong preference, i.e. two successive mis-predictions

Exercise

- Consider a loop branch (inner loop) that is taken nine times in a row, then is not taken once. What is the prediction accuracy for this branch?
  - Assume 2-bit history
  - Assume we initialize to strong taken – top left state

Pipeline Performance

- Ideal pipelined performance: $CPI_{ideal}=1$
- Hazards introduce additional stalls
  - $CPI_{pipelined}=CPI_{ideal}$+Average stall cycles per instruction
- Example
  - Instruction mix:
    - load 25%, store 10%, branches 11%, jumps 2%, ALU 52%
  - Half of the load followed immediately by an instruction that uses the result (in EX)
  - Branch delay on misprediction is 1 cycle and 1/4 of the branches are mispredicted (no delay if correct prediction)
  - Jumps always pay 1 cycle of delay
  - What is the average CPI?

Pipeline Performance Example

- Instruction mix:
  - load 25%, store 10%, branches 11%, jumps 2%, ALU 52%
- Hazard 1: Half of the load followed immediately by an instruction that uses the result (in EX)
  - Load-use hazard: 1 cycle delay assuming forwarding available
- Average stall cycles introduced per instruction
  - = 1 cycle * 25% * 50%
  - = 0.125 (cycle per instruction)

Pipeline Performance Example

- Instruction mix:
  - load 25%, store 10%, branches 11%, jumps 2%, ALU 52%
- Hazard 2: Branch delay on mis-prediciton is 1 cycle and 1/4 of the branches are mis-predicted (no delay if correct prediction)
- Average stall cycles introduced per instruction
  - = 1 cycle * 11% * (1/4)
  - = 0.0275 (cycle per instruction)

Pipeline Performance Example

- Instruction mix:
  - load 25%, store 10%, branches 11%, jumps 2%, ALU 52%
- Hazard 3: Jumps always pay 1 cycle of delay
- Average stall cycles introduced per instruction
  - = 1 cycle * 2%
  - = 0.02 (cycle)

Pipeline Performance

- Ideal pipelined performance: $CPI_{ideal}=1$
- Hazards introduce additional stalls
  - $CPI_{pipelined}=CPI_{ideal}$+Average stall cycles per instruction
- Example
  - Instruction mix:
    - load 25%, store 10%, branches 11%, jumps 2%, ALU 52%
  - Half of the load followed immediately by an instruction that uses the result +0.125
  - Branch delay on misprediciton is 1 cycle and 1/4 of the branches are mispredicted (no delay if correct prediction) +0.0275
  - Jumps always pay 1 cycle of delay +0.02
  - What is the average CPI? 1+0.125+0.0275+0.02 = 1.1725

Roadmap

- Resolving hazards
  - Data hazards: forwarding
  - Stalling the pipeline
  - Control hazards: prediction
- Advanced ILP techniques
  - Static approaches
    - Reordering, multiple issue, loop unrolling, …
  - Dynamic scheduling and speculation
- Concluding remarks

Instruction-Level Parallelism (ILP)

- Pipelining: executing multiple instructions in parallel
- More approaches to increase ILP
  - Deeper pipeline
    - Less work per stage ⇒ shorter clock cycle
    - Potential speedup (number of pipeline stages) increases
  - Multiple issue
    - Add more hardware to build multiple pipelines
    - Start multiple instructions per clock cycle
    - Dependencies enforce restrictions
  - Reordering instructions to cover pipeline stalls
    - Dependencies enforce restrictions

Data Dependences

- Flow dependences - read after write

```
x := 4 + a;
…
y := x + 1;
```

- Anti-dependences - write after read

```
y := x + 1;
…
x := 4 + a;
```

- Output dependences - write after write

```
x := 4 + a;
…
x := y + 1;
```

- The latter two can be eliminated by renaming
  - No value flows (also called name dependences)
  - Flow dependences are also called true dependences

Basic rule: If there is data dependence between two instructions, we must keep the execution order.

Roadmap

- Resolving hazards
  - Data hazards: forwarding
  - Stalling the pipeline
  - Control hazards: prediction
- Advanced ILP techniques
  - Static approaches ←
    - Reordering, multiple issue, loop unrolling, …
  - Dynamic scheduling and speculation
- Concluding remarks

Code Reordering to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction

```
I0: lw  $t1, 0($t0)
I1: lw  $t2, 4($t0)
I2: add $t3, $t1, $t2
I3: sw  $t3, 12($t0)
I4: lw  $t4, 8($t0)
I5: add $t5, $t1, $t4
I6: sw  $t5, 16($t0)
```

(reg, I_producer, I_consumer)

  - First: identify true dependences (reordering constraints)
    - ($t1, I0, I2), ($t1, I0, I5), ($t2, I1, I2), ($t3, I2, I3), ($t4, I4, I5), ($t5, I5, I6)

Code Reordering to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction

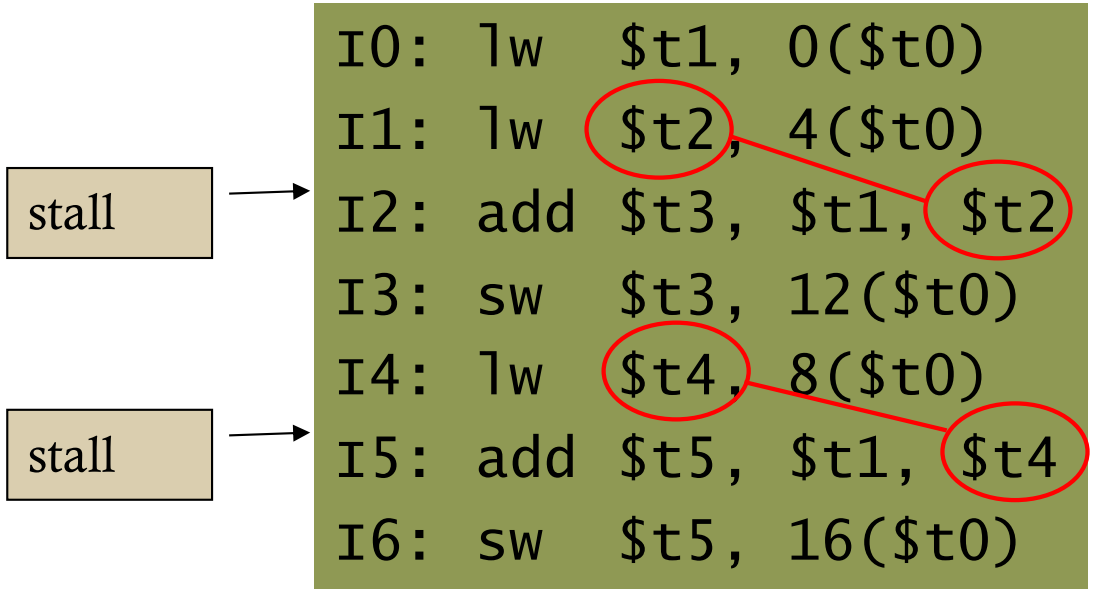
($t1, I0, I2), ($t2, I1, I2),
($t3, I2, I3), ($t4, I4, I5),
($t5, I5, I6), ($t1, I0, I5)

```
I0: lw  $t1, 0($t0)
I1: lw  $t2, 4($t0)
I2: add $t3, $t1, $t2
I3: sw  $t3, 12($t0)
I4: lw  $t4, 8($t0)
I5: add $t5, $t1, $t4
I6: sw  $t5, 16($t0)
```

  - First: identify true dependences
  - Then: identify load-use hazards (assume forwarding will take care of all other data hazards)
    - E.g. sw needs additional support to accept forwarded $rt

Code Reordering to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction

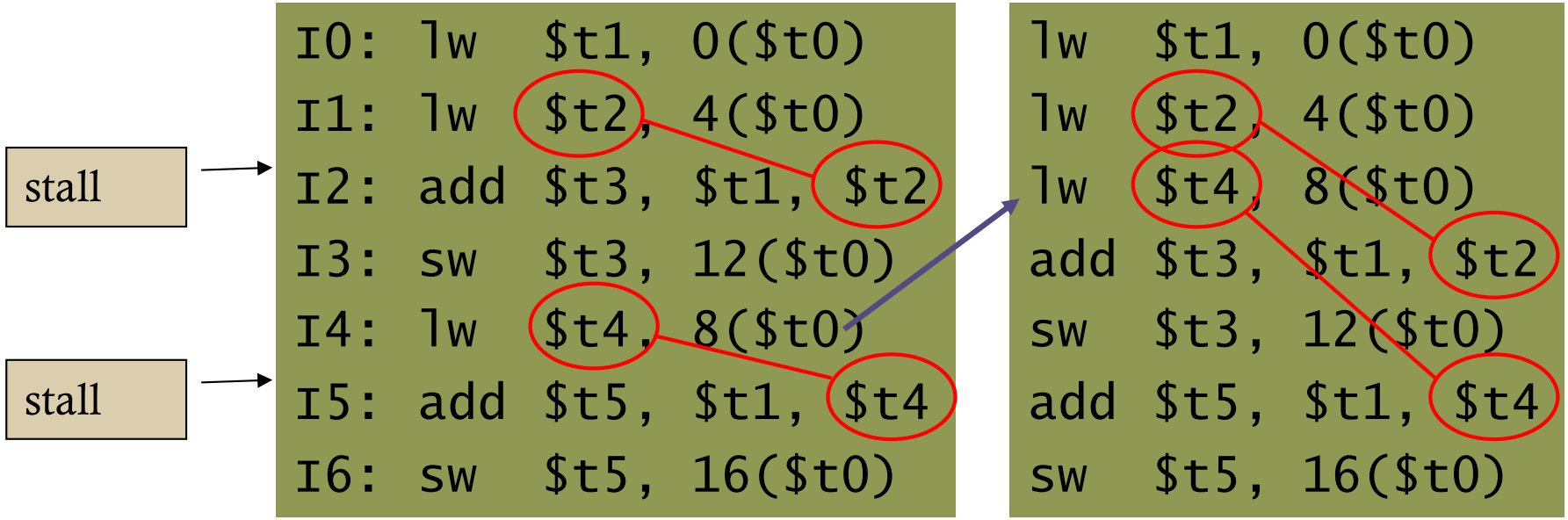


($t1, I0, I2), ($t2, I1, I2),
($t3, I2, I3), ($t4, I4, I5),
($t5, I5, I6), ($t1, I0, I5)

13 cycles to complete: 9 cycles to start last sw; 4 more cycles to finish

Code Reordering to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction



13 cycles to complete: 9 cycles to start last sw; 4 more cycles to finish

11 cycles to complete: start one instruction @ every cycle