Instruction scheduling

Based on slides by Ilhyun Kim and Mikko Lipasti

Schedule of things to do

- HW5 posted.
  - Due on 4/23 (last day of class) @10pm – 24 hours late with only 5% off.
- MS3 Today
- Project is due on Saturday 4/20 at 9pm.
  - Last synth job can still be running, but don't rely on it.
  - No code changes after this.
- Oral and written reports are due on Tuesday 4/23.
  - Brief directions for both posted shortly.
  - Final written report is due at 9pm on Piazza
  - Oral reports are during the day.
  - Will watch at least 2 other talks.
    - Can go to all if you wish.

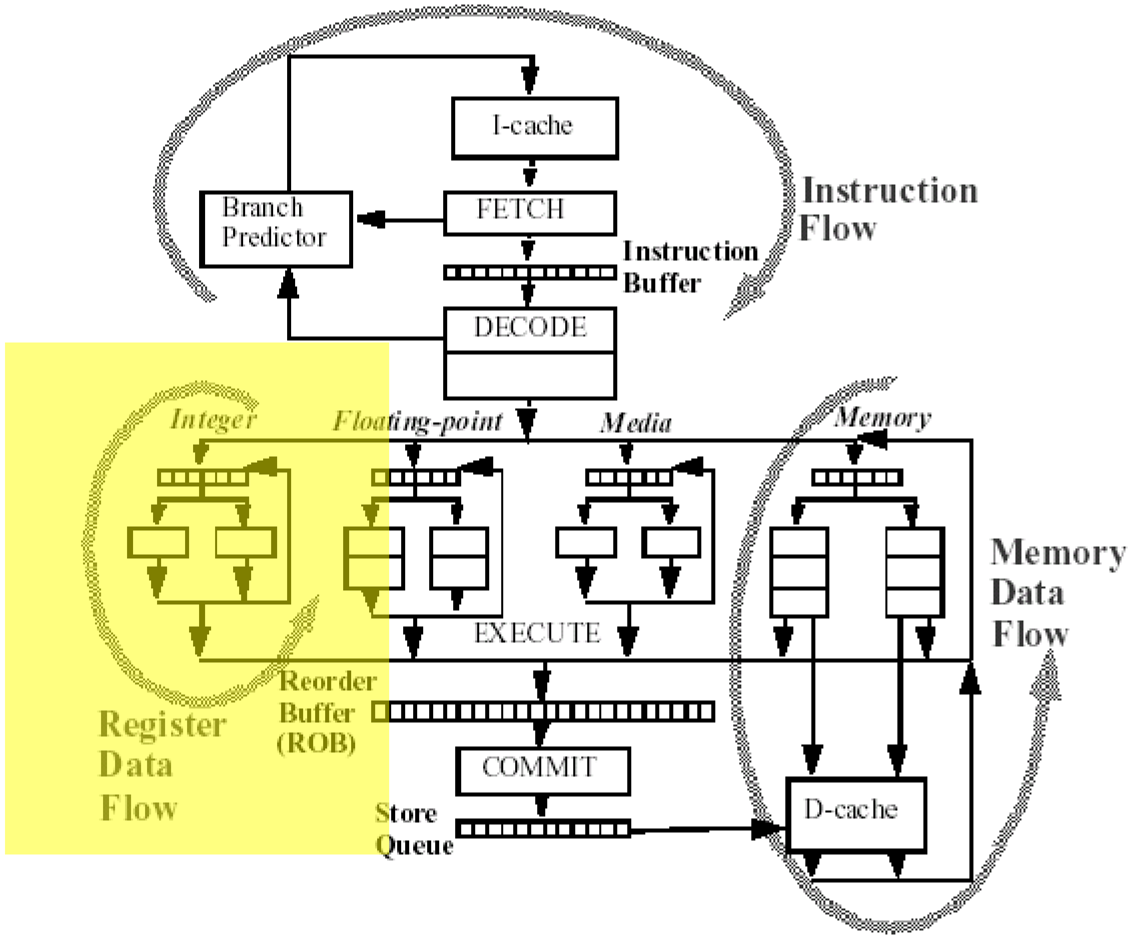
Lecture (and review) schedule

- 4/11: (today) Instruction scheduling
- 4/16: (Tuesday) No class (work on projects)
- 4/18: (Thursday) Exam Q&A in class
- 4/23: (Tuesday) Project presentations
- 4/24: (Wednesday) Exam Q&A (what time?)
- 4/26: (Friday) Final exam 10:30-12:30

Today

- Instruction scheduling overview
  - Scheduling atomicity
  - Speculative scheduling
  - Scheduling recovery
- Other neat ideas…
- Reading list

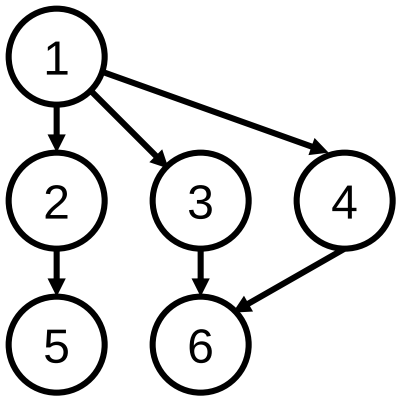
Register Dataflow



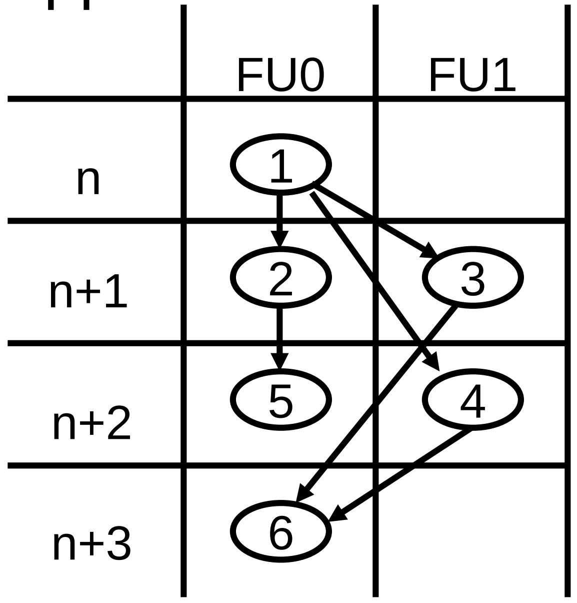
Instruction scheduling

- A process of mapping a series of instructions into execution resources
  - Decides when and where an instruction is executed

- Data dependence graph



- Mapped to two FUs

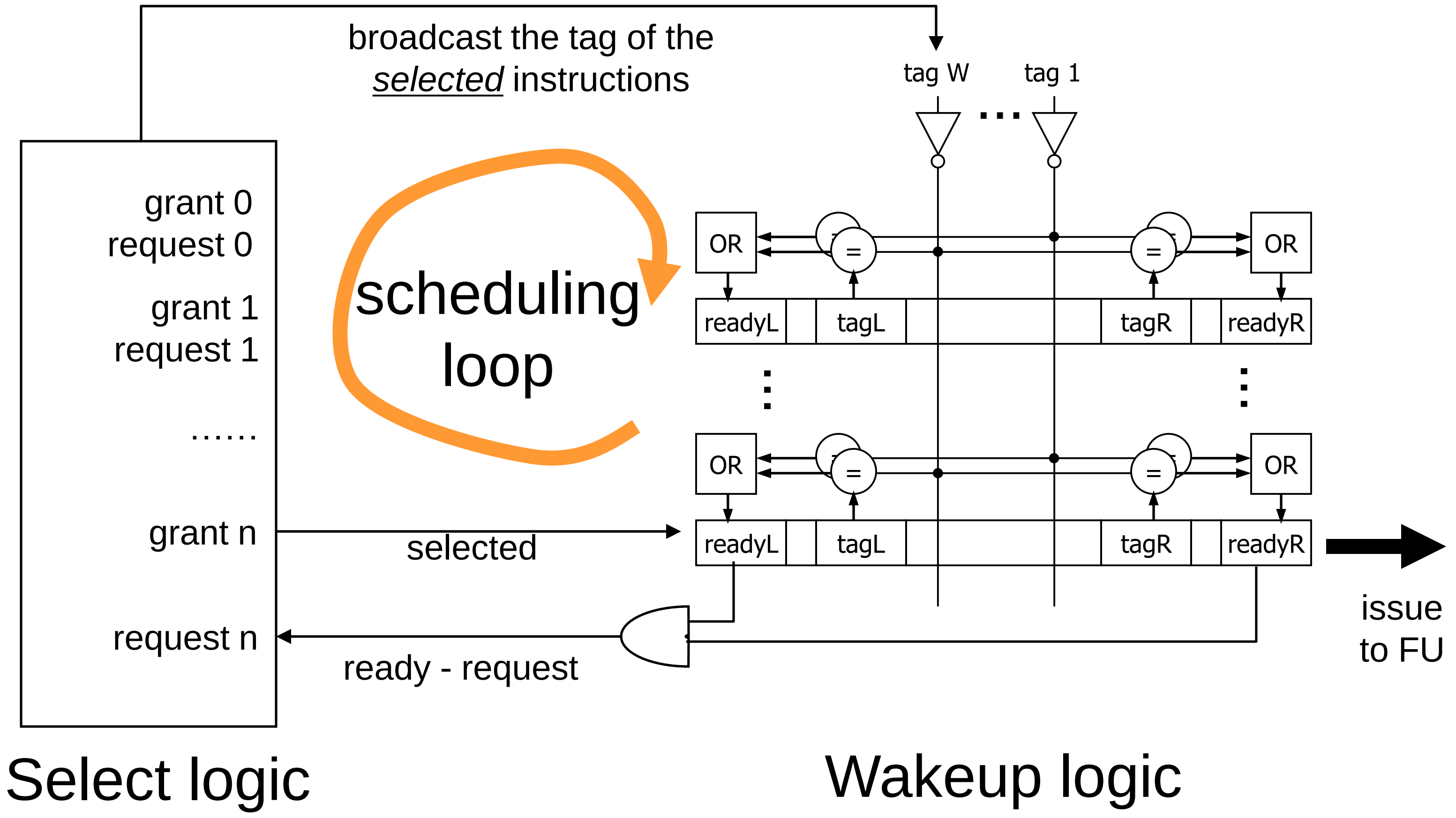


Instruction scheduling

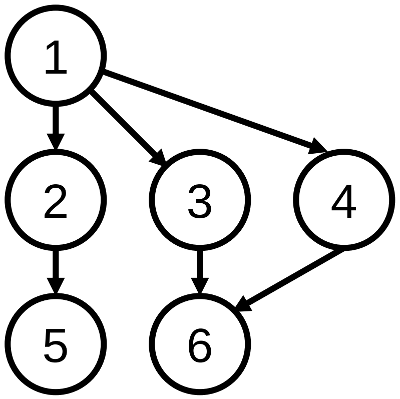
- A set of wakeup and select operations
  - Wakeup
    - Broadcasts the tags of parent instructions selected
    - Dependent instruction gets matching tags, determines if source operands are ready
    - *<u>Resolves</u> true data dependences*
  - Select
    - Picks instructions to issue among a pool of ready instructions
    - *<u>Resolves</u> resource conflicts*
      - Issue bandwidth
      - Limited number of functional units / memory ports

Scheduling loop

- Basic wakeup and select operations



Wakeup and Select

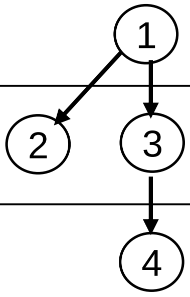


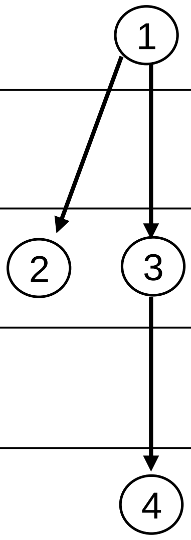
	FU0	FU1	Ready inst to issue	Wakeup / select
n	1		1	Select 1 Wakeup 2,3,4
n+1	2	3	2, 3, 4	Select 2, 3 Wakeup 5, 6
n+2	5	4	4, 5	Select 4, 5 Wakeup 6
n+3	6		6	Select 6

Scheduling Atomicity

- If we want to pipeline selection logic, we will latch the selection decision (it becomes a pipeline stage)
  - So we can’t wake up the next guy until the cycle after we are selected.

cycle	**Atomic scheduling**	**Non-Atomic 2-cycle scheduling**
n	select 1 wakeup 2, 3	select 1
n+1	Select 2, 3 wakeup 4	wakeup 2, 3
n+2	Select 4	select 2, 3
n+3		wakeup 4
n+4		select 4



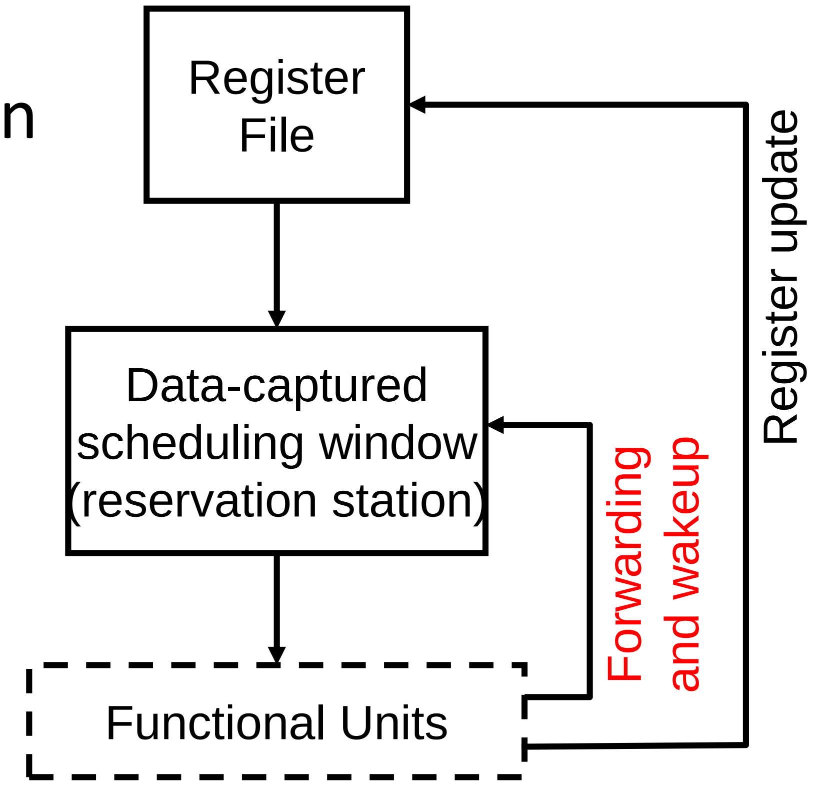


Implication of scheduling atomicity

- Pipelining is a standard way to improve clock frequency
- Hard to pipeline instruction scheduling logic without losing ILP
  - ~10% IPC loss in 2-cycle scheduling
  - ~19% IPC loss in 3-cycle scheduling
- A major obstacle to building high-frequency microprocessors

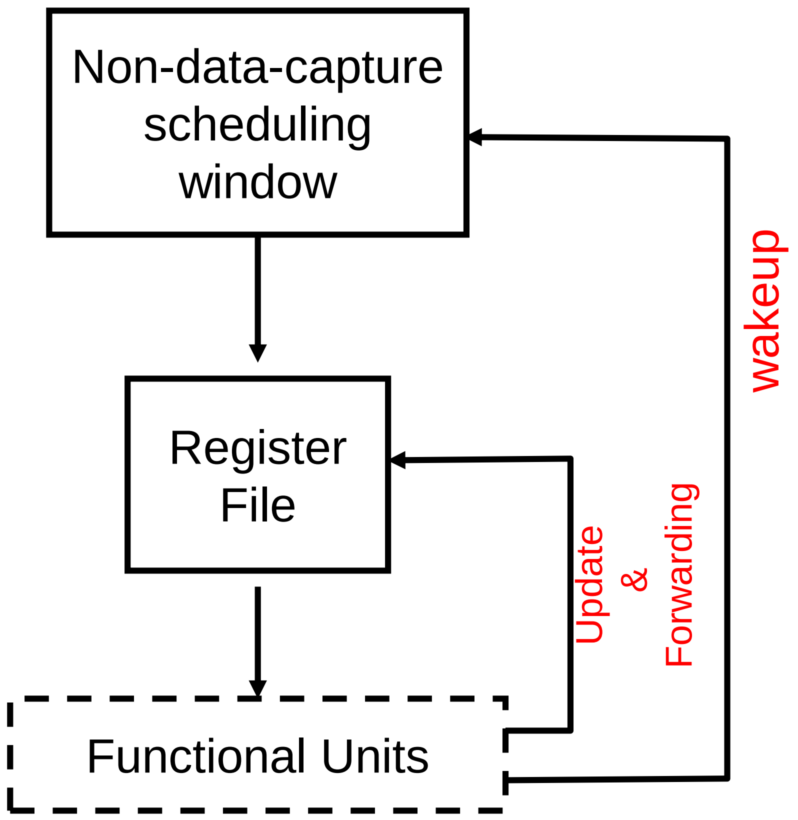
Scheduler Designs

- Data-Capture Scheduler
  - Keep the most recent register value in reservation stations
  - Data forwarding and wakeup are combined to some extent
    - Early tag broadcast decouples this to some extent of course.



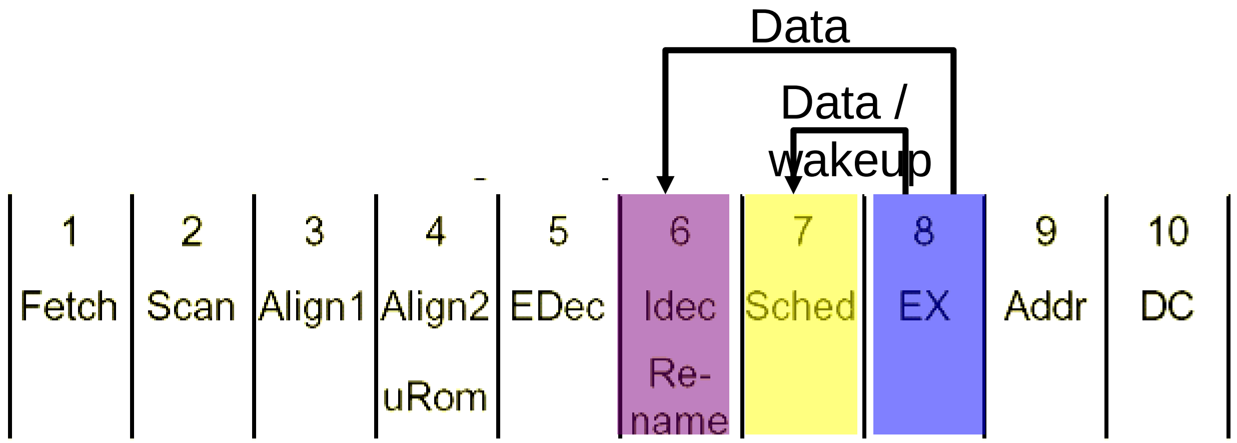
Scheduler Designs

- Non-Data-Capture Scheduler
  - Keep the most recent register value in RF (physical registers)
  - Data forwarding and wakeup are cleanly decoupled

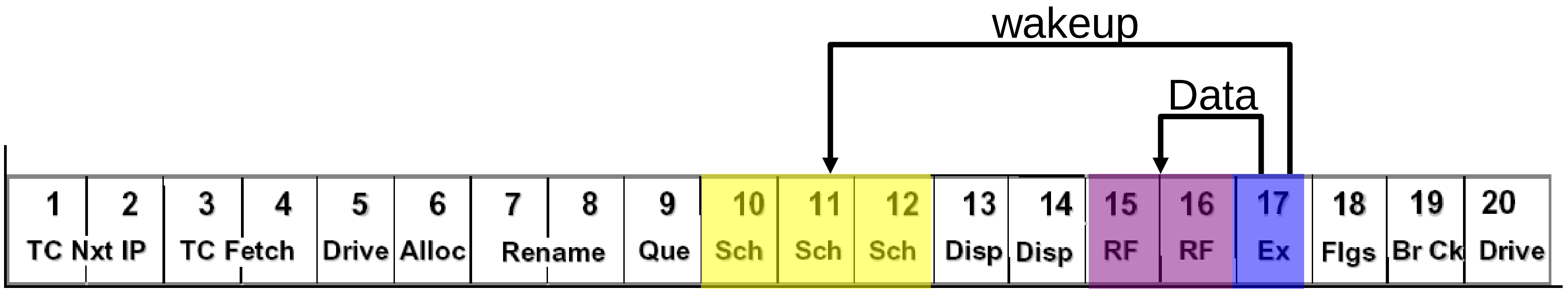


Mapping to pipeline stages

- AMD K7 (data-capture)

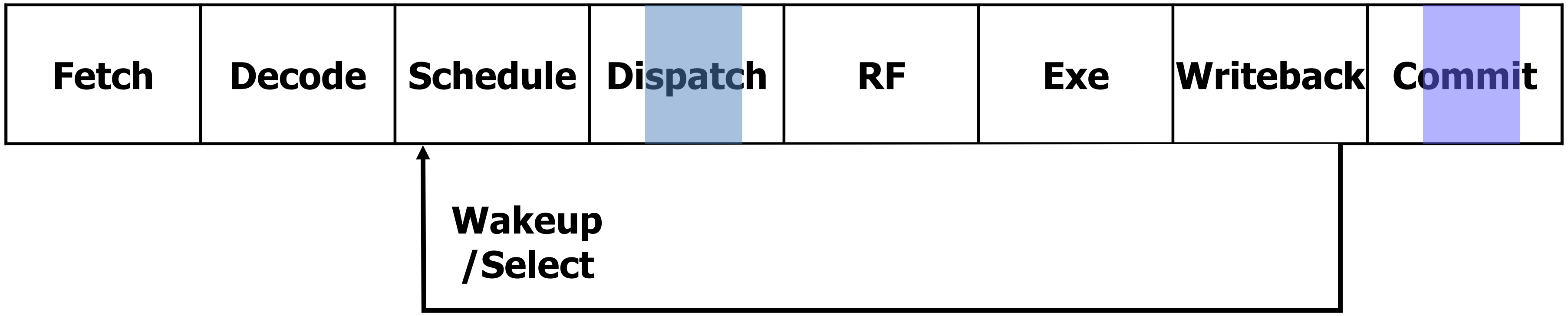


- Pentium 4 (non-data-capture)



Scheduling atomicity & non-data-capture scheduler

- Multi-cycle scheduling loop



- Scheduling atomicity is not maintained
  - Separated by extra pipeline stages (Disp, RF)
  - Unable to issue dependent instructions consecutively

➔ solution: speculative scheduling

Speculative Scheduling

- Speculatively wakeup dependent instructions even before the parent instruction ***starts*** execution
  - Keep the scheduling loop within a single clock cycle

- But, nobody knows what will happen in the future

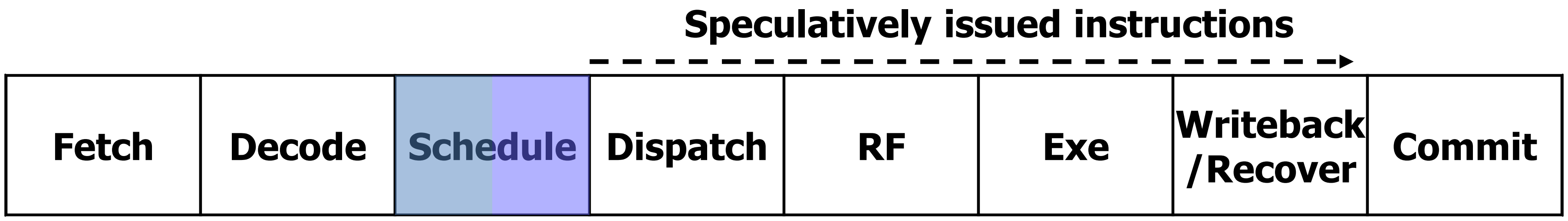
- Source of uncertainty in instruction scheduling: loads
  - Cache hit / miss
  - Store-to-load aliasing

  ➔ eventually affects timing decisions

- Scheduler assumes that all types of instructions have pre-determined fixed latencies
  - Load instructions are assumed to have a common case (over 90% in general) $DL1 hit latency
  - If incorrect, subsequent (dependent) instructions are replayed

Speculative Scheduling

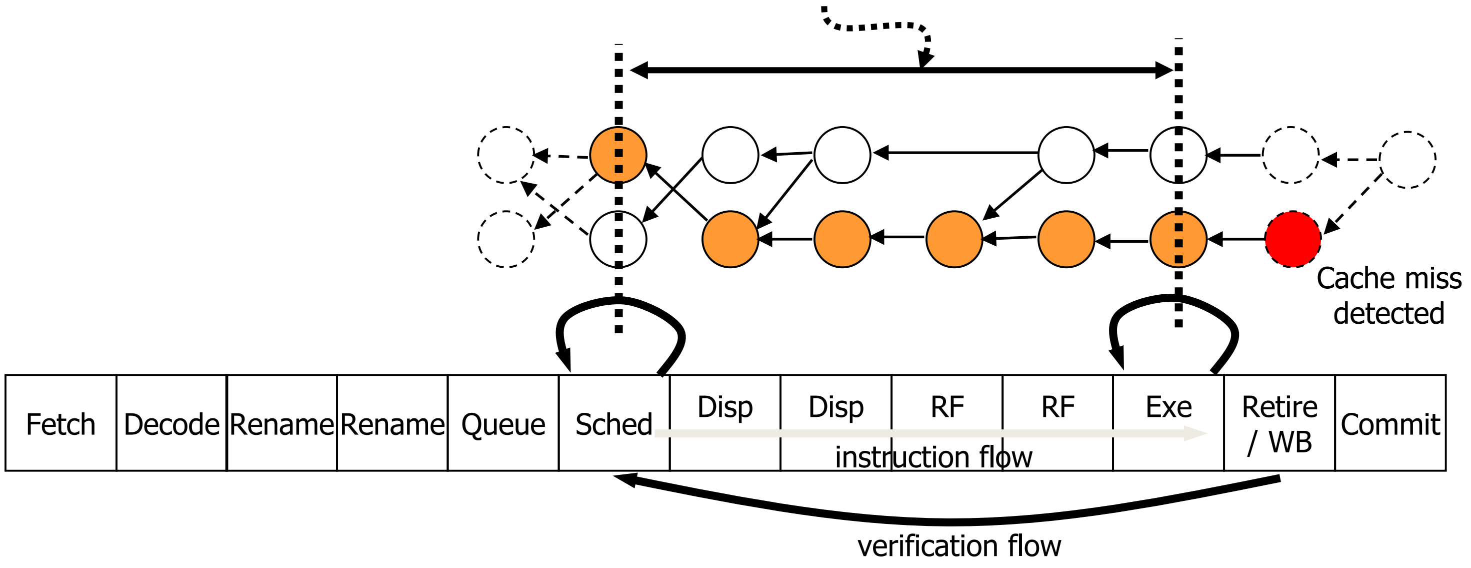
- Overview



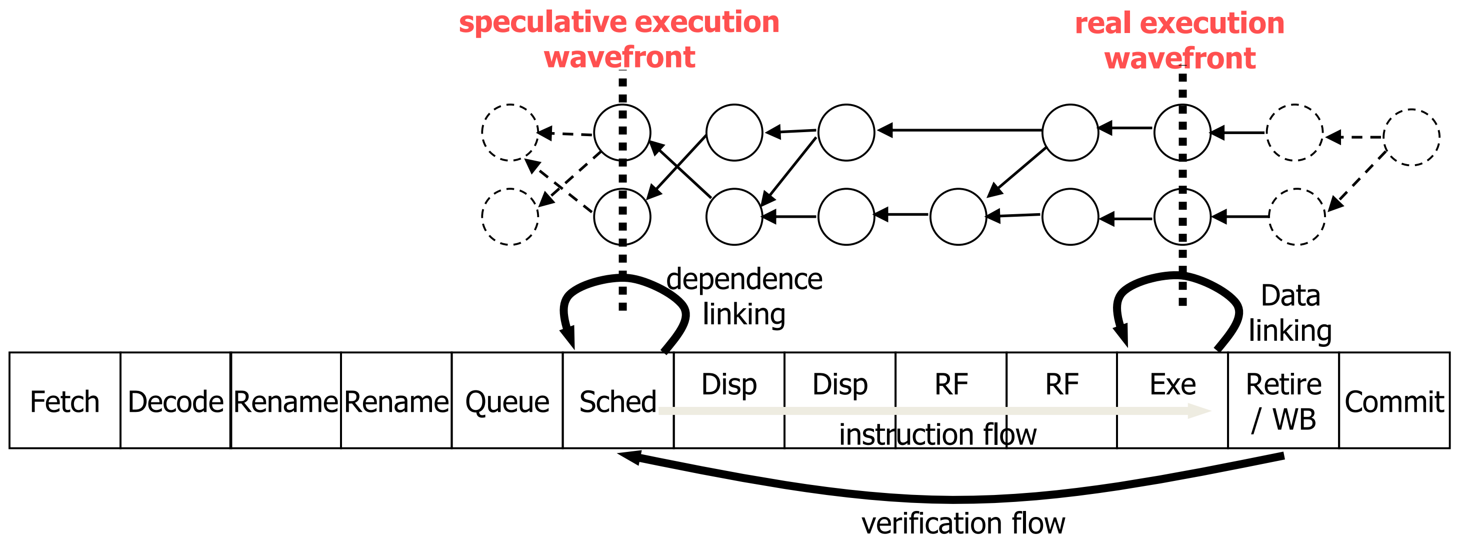
- Unlike the original Tomasulo's algorithm
  - Instructions are scheduled BEFORE actual execution occurs
  - Assumes instructions have pre-determined fixed latencies
    - ALU operations: fixed latency
    - Load operations: assumes $DL1 latency (common case)

Scheduling replay

- Speculation needs verification / recovery
  - There's no free lunch
- If the actual load latency is longer (i.e. cache miss) than what was speculated
  - Best solution (disregarding complexity): replay data-dependent instructions issued under *load shadow*

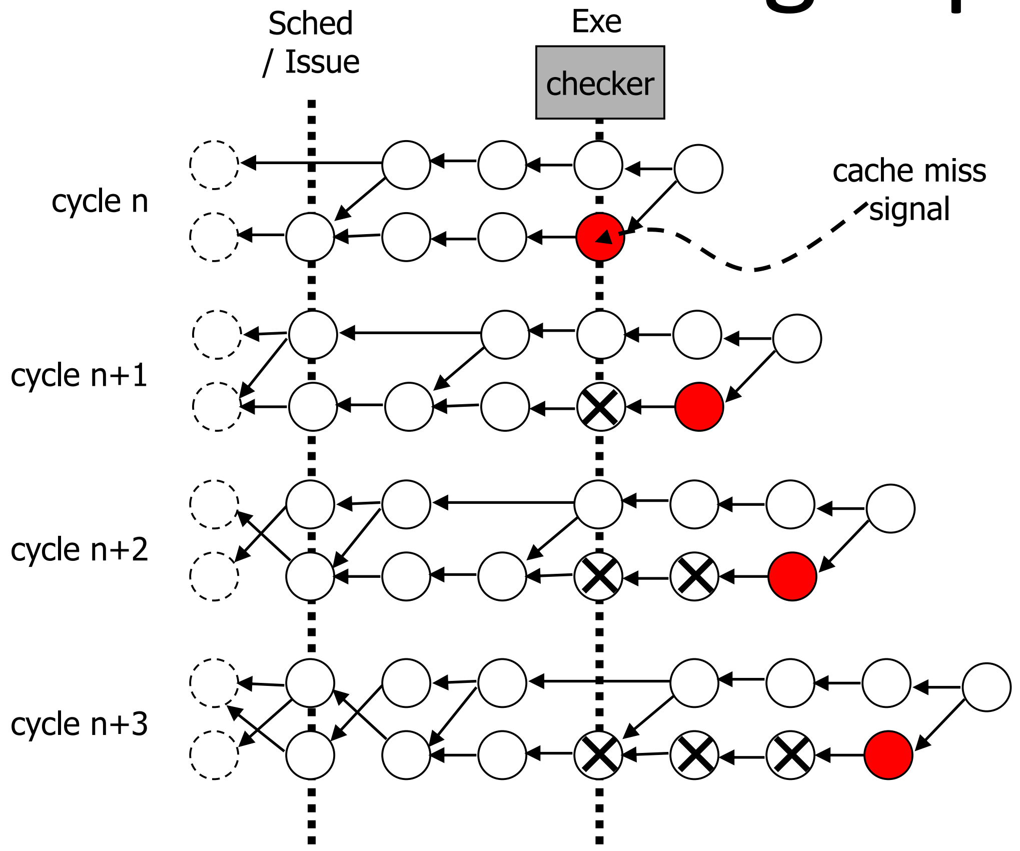


Wavefront propagation



- Speculative execution wavefront
  - speculative image of execution (from scheduler's perspective)
- Both wavefront propagates along dependence edges at the same rate (1 level / cycle)
  - the real wavefront runs behind the speculative wavefront
- The load resolution loop delay complicates the recovery process
  - scheduling miss is notified a couple of clock cycles later after issue

Issues in scheduling replay



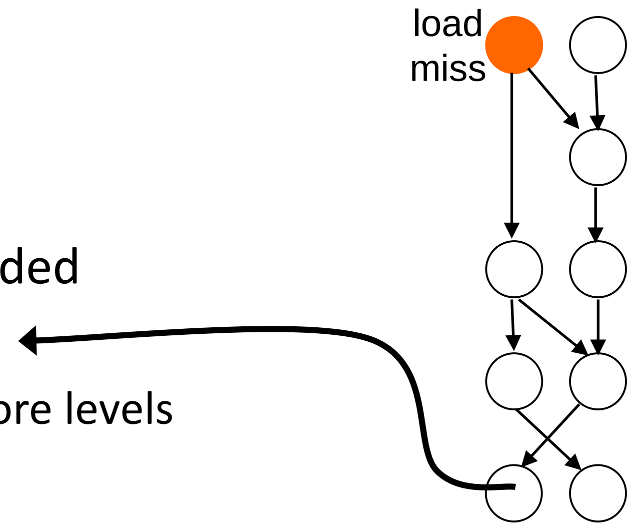
- Cannot stop speculative wavefront propagation
  - Both wavefronts propagate at the same rate
  - Dependent instructions are unnecessarily issued under load misses

Requirements of scheduling replay

- Propagation of recovery status should be faster than speculative wavefront propagation
- Recovery should be performed on the transitive closure of dependent instructions

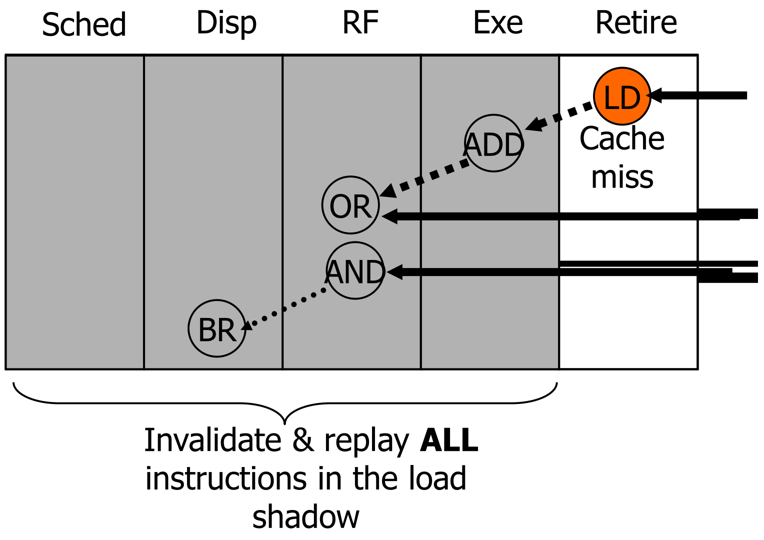
- Conditions for ideal scheduling replay
  - All mis-scheduled dependent instructions are invalidated instantly
  - Independent instructions are unaffected

- Multiple levels of dependence tracking are needed
  - e.g. Am I dependent on the current cache miss?
  - Longer load resolution loop delay → tracking more levels

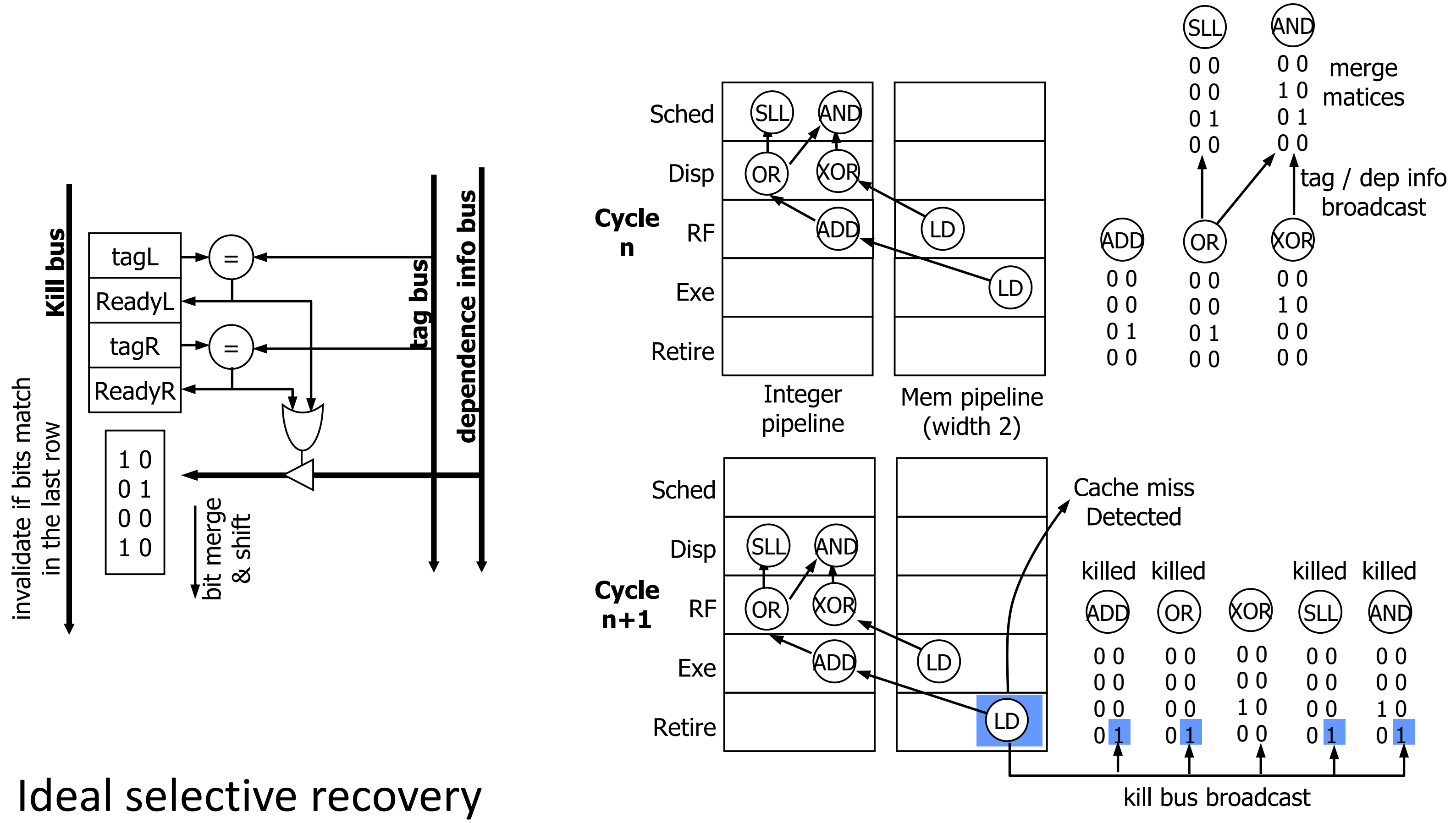


Scheduling replay schemes

- Alpha 21264: Non-selective replay
  - Replays all dependent and independent instructions issued under load shadow
  - Analogous to squashing recovery in branch misprediction
  - Simple but high performance penalty
    - Independent instructions are unnecessarily replayed



Position-based selective replay



- Ideal selective recovery
  - replay dependent instructions only
- Dependence tracking is managed in a matrix form
  - Column: load issue slot, row: pipeline stages

We could also do something more radical

- Greatly simplify scheduling in some way.

Another scheduling idea: Grandparents

- Schedule based on grandparents

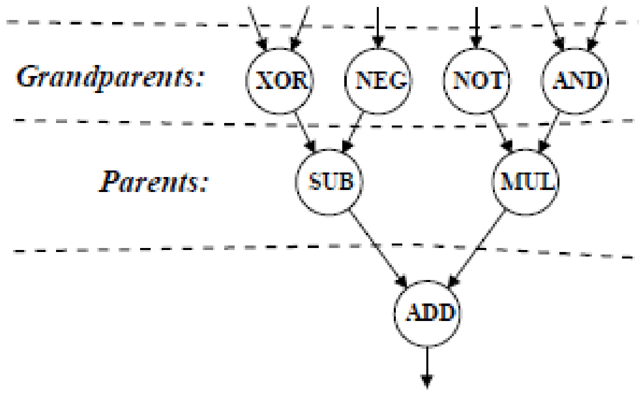
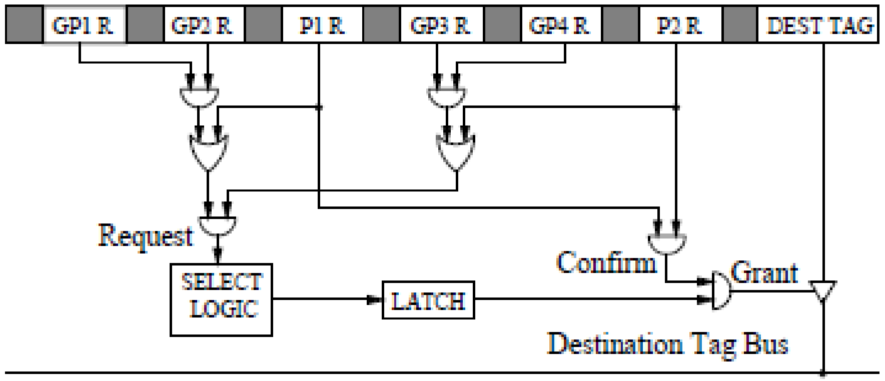


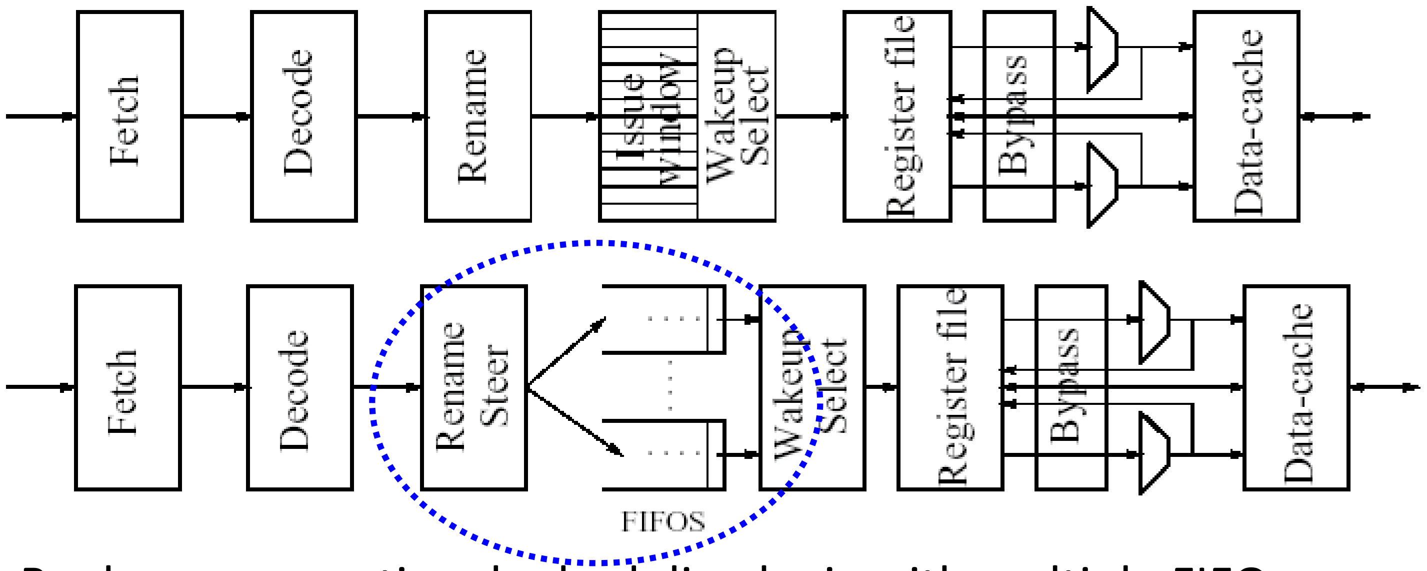
Figure 2. Example Data Flow Graph



J. Stark, M.D. Brown, and Y.N. Patt. "On pipelining dynamic instruction scheduling logic," ISCA 2000

Low-complexity scheduling techniques

- FIFO (Palacharla, Jouppi, Smith, 1996)

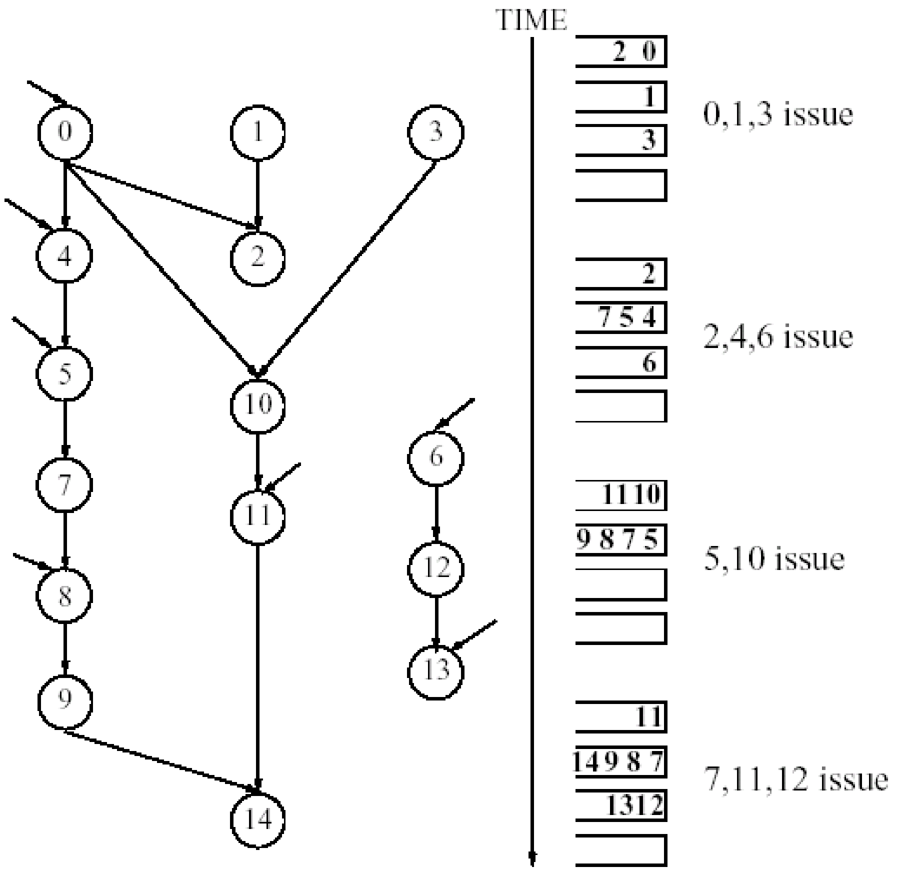


  – Replaces conventional scheduling logic with multiple FIFOs
    - Steering logic puts instructions into different FIFOs considering dependences
    - A FIFO contains a chain of dependent instructions
    - Only the head instructions are considered for issue

FIFO (cont’d)

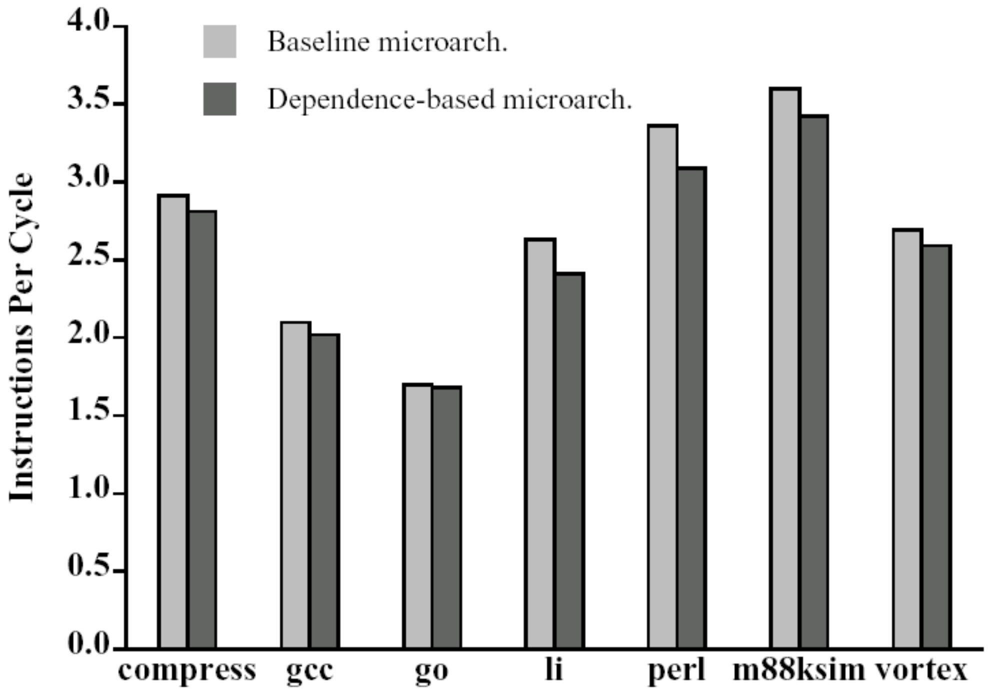
- Scheduling example

```
 0: addu $18,$0,$2
 1: addiu $2,$0,-1
 2: beq $18,$2,L2
 3: lw $4,-32768($28)
 4: sllv $2,$18,$20
 5: xor $16,$2,$19
 6: lw $3,-32676($28)
 7: sll $2,$16,0x2
 8: addu $2,$2,$23
 9: lw $2,0($2)
10: sllv $4,$18,$4
11: addu $17,$4,$19
12: addiu $3,$3,1
13: sw $3,-32676($28)
14: beq $2,$17,L3
```



FIFO (cont'd)

- Performance



- Comparable performance to the conventional scheduling
- Reduced scheduling logic complexity
- Many related papers on *clustered microarchitecture*
- Can in-order clusters provide high performance?

Key Challenge: MLP (Memory-Level Parallelism)

- Tolerate/overlap memory latency
  - Once first miss is encountered, find another one
- Naïve solution
  - Implement a very large ROB, LSQ
  - Power/area/delay make this infeasible
- Build *virtual* instruction window
  - How to do this?

Check point

- Key notion is we need to be able to recover when we get a mis-speculation (or exception or other nuke situation)
  - How about just storing a check point every X instructions (say 100).
    - If there is a nuke, back up to check point and move forward with either
      - Knowledge of issue (predict correctly this time) OR
      - Carefully (in-order?).
    - Don’t let stores write to memory until get to next check point.

Sources and Further Reading

- I. Kim and M. Lipasti, “Understanding Scheduling Replay Schemes,” in Proceedings of the 10th International Symposium on High-performance Computer Architecture (HPCA-10), February 2004.
- Srikanth Srinivasan, Ravi Rajwar, Haitham Akkary, Amit Gandhi, and Mike Upton, “Continual Flow Pipelines”, in Proceedings of ASPLOS 2004, October 2004.
- Ahmed S. Al-Zawawi, Vimal K. Reddy, Eric Rotenberg, Haitham H. Akkary, “Transparent Control Independence,” in Proceedings of ISCA-34, 2007.
- T. Shaw, M. Martin, A. Roth, “NoSQ: Store-Load Communication without a Store Queue, ” in Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, 2006.
- Andrew Hilton, Santosh Nagarakatte, Amir Roth, "iCFP: Tolerating All-Level Cache Misses in In-Order Processors," Proceedings of HPCA 2009.
- J. Stark, M.D. Brown, and Y.N. Patt. “On pipelining dynamic instruction scheduling logic,” ISCA 2000