

---

# Pipelining, Superscalar and Out-of-order

Courtesy: A. Moshovos. Full PPT available at:  
[www.eecg.toronto.edu/~moshovos/ACA06/lecturenotes/005-superscalar.ppt](http://www.eecg.toronto.edu/~moshovos/ACA06/lecturenotes/005-superscalar.ppt)

Slides adapted by: Dr Sparsh Mittal

---

# Sequential Semantics - Review

---

- Instructions appear as if they executed:
    - In the order they appear in the program
    - One after the other
  - **Pipelining:** Partial Overlap of Consecutive Instructions
    - Initiate one instruction per cycle
    - Subsequent instructions overlap partially
    - Commit one instruction per cycle
-

# Can we do better than pipelining?

---

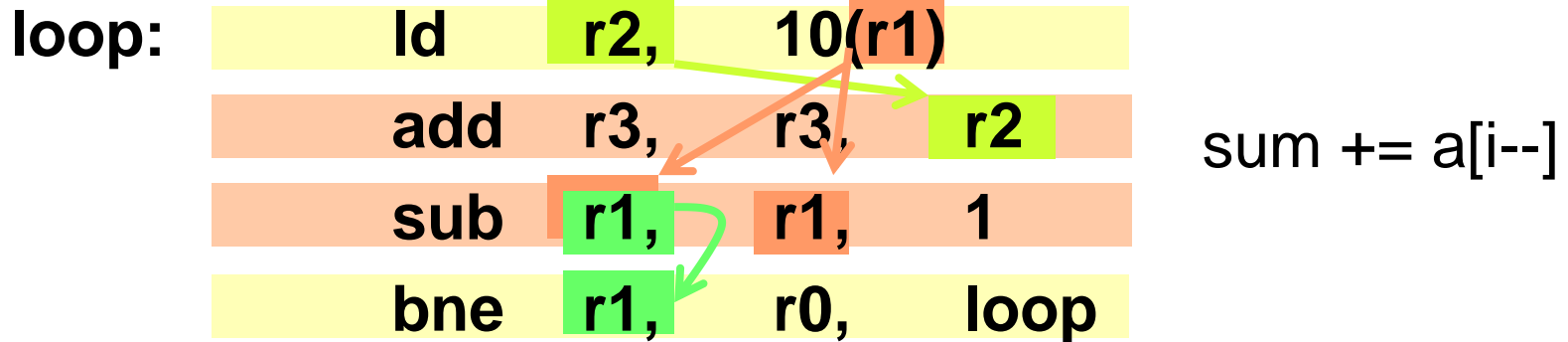
We will try to answer this question by studying execution of:

`sum += a[i--]`

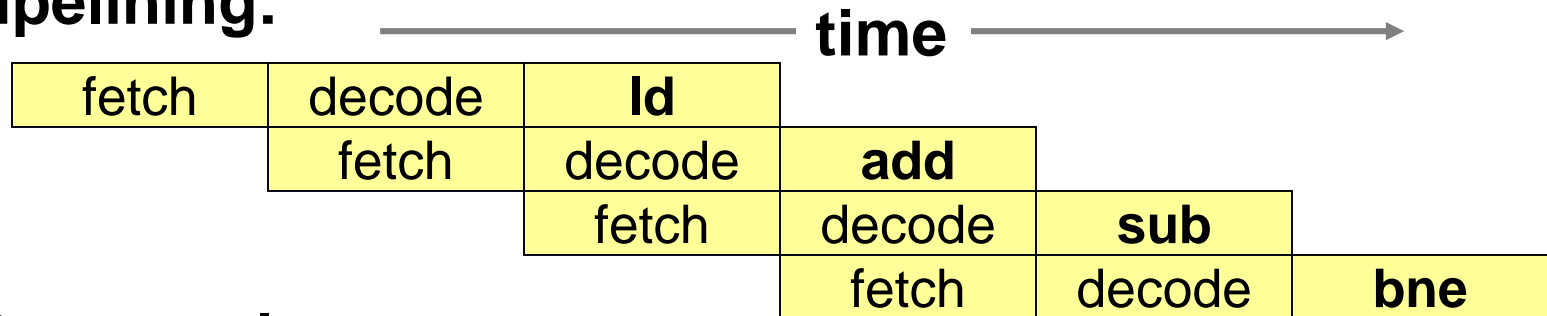
We will explore **superscalar** execution.

---

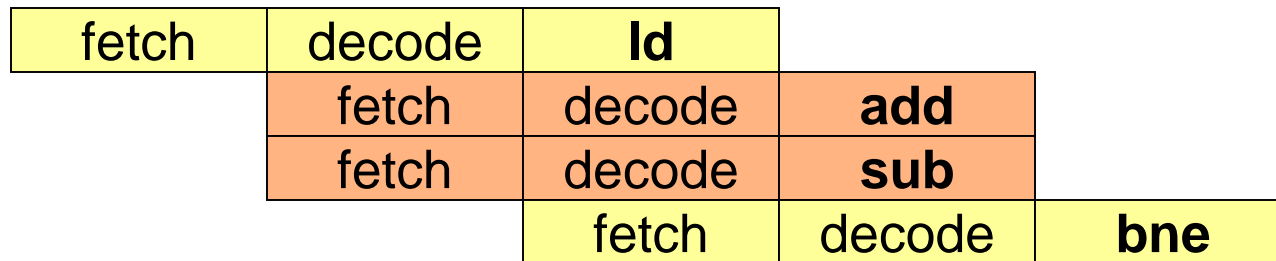
# Can we do better than pipelining?



Pipelining:



Superscalar:



## Superscalar - In-order (initial def.)

---

- Two or more ***consecutive*** instructions (in the original program order) can execute in parallel
  - Is this much better than pipelining?
    - What if all instructions were dependent?
      - Superscalar buys us nothing
  - Increasingly Complex with ***degree of superscalarity***
    - 2-way, 3-way, ..., n-way
-

# Implications of Superscalar

---

- **Need to multiport some structures**
    - Register File
      - Multiple Reads and Writes per cycle
    - Register Availability Vector
      - Multiple Reads and Writes per cycle
        - From Decode and Commit
        - Also need to worry about WAR and WAW
  - **Resource tracking**
    - Additional issue conditions
-

# Preserving Sequential Semantics

---

- **In principle, Superscalar not much different than pipelining**
  - **Program order is preserved in the pipeline**
  - **Some instructions proceed in parallel**
    - But order is clearly defined
-

# Superscalar vs. Pipelining

---

- **In principle they are orthogonal. We can have**
    - Superscalar non-pipelined machine
    - Pipelined non-superscalar
    - Superscalar and Pipelined (common)
-



# Out-of-Order Execution

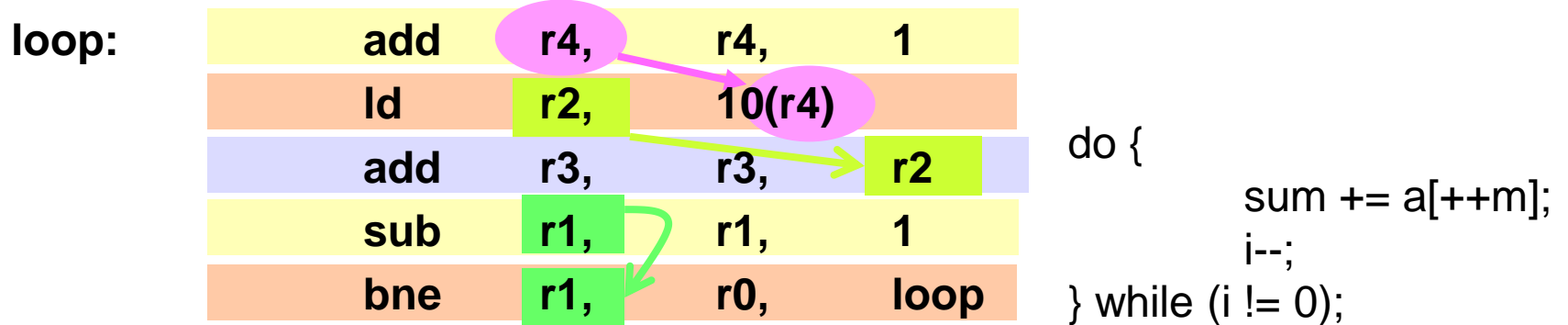
---

- **Also known as dynamic scheduling**
  - Compilers do static scheduling
- **We will start by considering register only**
  - Register interface helps a lot

Following code will be used as an example:

```
do {  
    sum += a[++m];  
    i--;  
} while (i != 0);
```

# Beyond Superscalar Execution



## Superscalar:

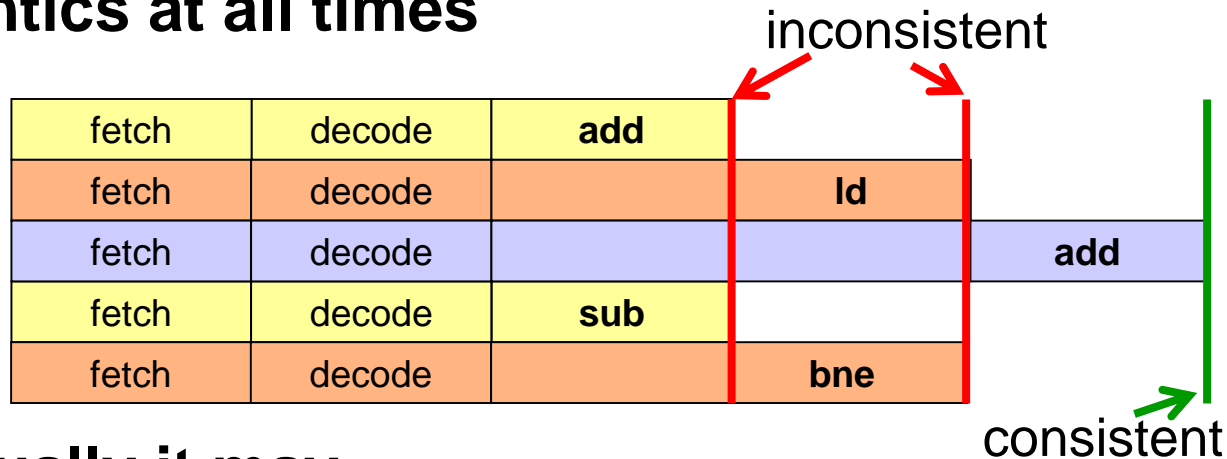
fetch	decode	add			
fetch	decode		ld		
fetch	decode			add	
fetch	decode			sub	
fetch	decode				bne

## out-of-order

fetch	decode	add		
fetch	decode		ld	
fetch	decode			add
fetch	decode	sub		
fetch	decode		bne	

# Sequential Semantics?

- **Execution does NOT adhere to sequential semantics at all times**



- **Eventually it may**
- **Challenges of Out-of-Order Exec.: Imprecise interrupts**
  - On interrupt some instr. committed some not
  - software will have to figure out what is going on
  - Makes debugging and programming difficult

# Out-of-Order vs. Pipelining and Superscalar

---

- **Definition:** *two or more instructions can execute in **any** order if they have no dependences (RAW, WAW, WAR)*
  - **Is this better than pipelining or superscalar exec?**
    - If all are independent: not
    - if all dependent: not
    - Useful when programs have *some* parallelism
    - Superscalar: exploits **parallelism only when it is between adjacent instructions**
    - OoO exploits par. even when not adjacent
  - **OoO Orthogonal to pipelining and Superscalar**
-